



ET - RAD user manual

Indice

Parte I	ET Rad	5
1	ET RAD	5
Parte II	Delphi and Pascal overview	8
1	Object Pascal	8
2	Functions and procedures	10
3	Keywords	15
4	Expression	18
5	Operation instruction list	22
6	Integer and floating point numbers	25
7	Strings and characters	27
8	Case statements	31
9	Exception handling	33
10	Files	36
11	Dates and times	43
12	DataBase	47
Parte III	ET Feature overview	49
1	Language Features	49
	Managing includes modules	49
	3 MURI OP Include.....	50
	SysDataUtils include.....	50
	Pascal ET Syntax	52
	Overview	52
	Identifiers	52
	Assign statements	53
	Character strings.....	53
	Comments.....	53
	Variables	54
	Arrays	54
	If statement.....	55
	While statements.....	55
	Repeat statemets.....	55
	For statement.....	56
	Case statements	56
	Function and procedure declaration.....	57
	Supported types.....	57
	The TatSystemLibrary library.....	58
	Except - Starts the error trapping clause of a Try statement.....	60
Parte IV	Using ET RAD	62

1	Start New Project	62
2	Open Script Project	67
3	Save Script Project	68
4	Script Structure	71
5	ET RAD Environment	72
6	Debugging ET Script	74
7	Understanding the Script unit source	77
8	Standard tab GUI components	80
9	Additional tab GUI components	83
10	Win32 tab GUI components	84
11	Dialog tab GUI components	85
12	System tab GUI components	86
13	Open Dialog Component	87
14	Save Dialog Component	89
15	Font Dialog Component	91
16	Color Dialog Component	92
17	Print Dialog Component	95
18	Learn about: properties, events and ET RAD	97
19	Code Samples	99
	3 MURI OP	99
	3 MURI OP Interface	100
	Read 3Muri dataBase Access	105
	Replace function	107
	Read and Write INI Files	108
	Dll Call	110
	Split function	111
	DataSet	112
20	Sample Script	113

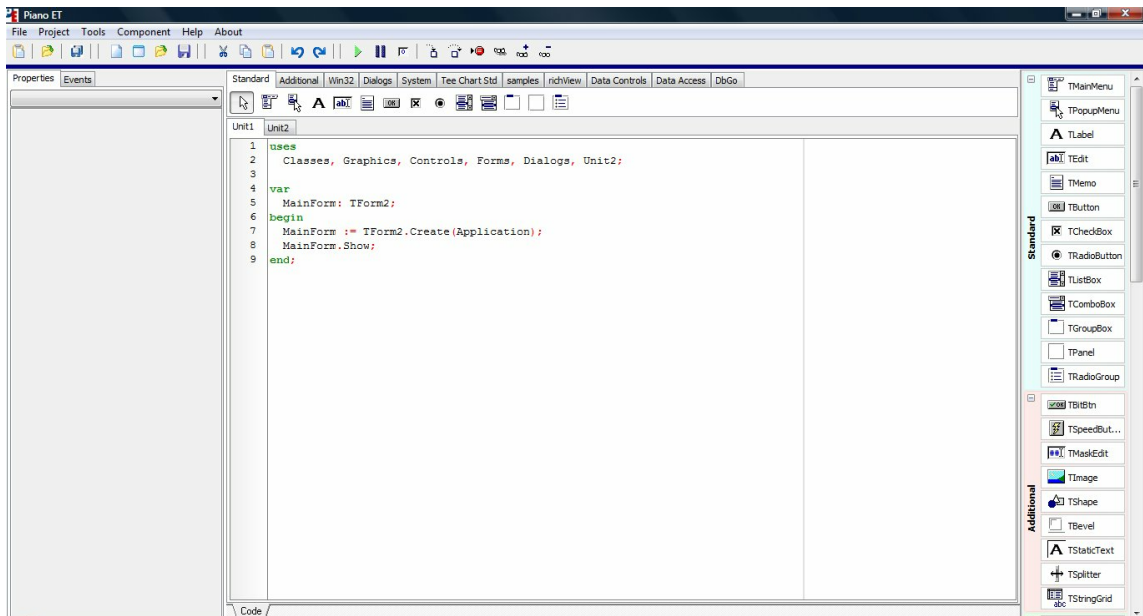
1 ET Rad

1.1 ET RAD

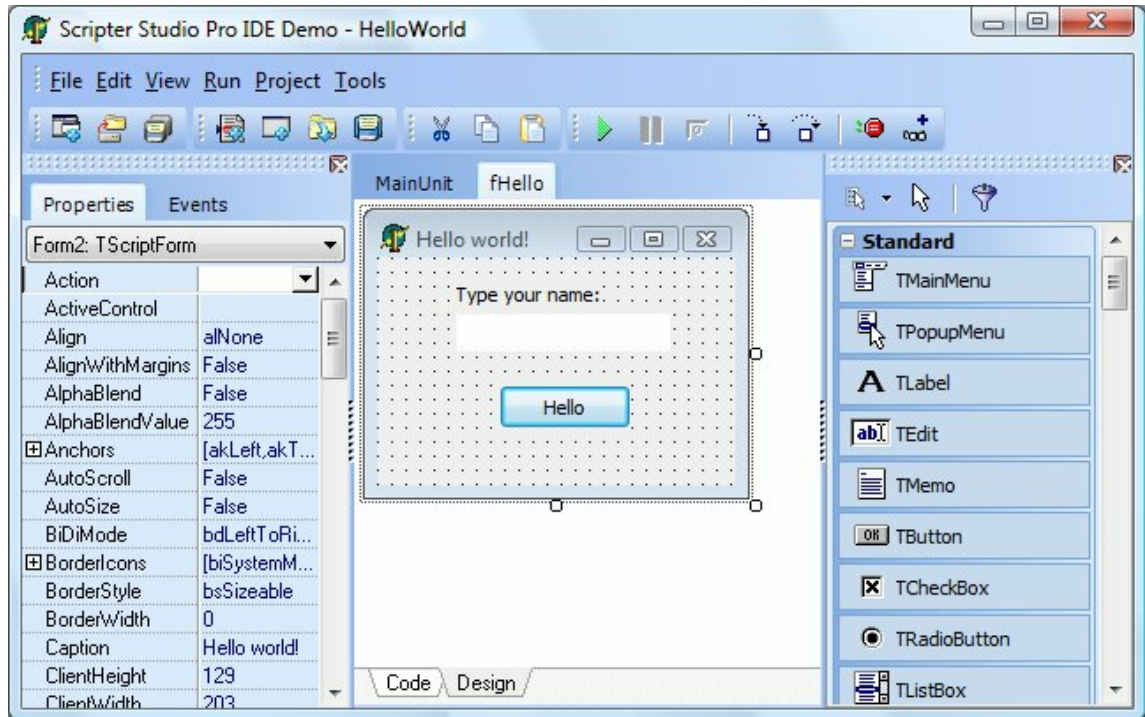
Navigation: ET Rad >

ET RAD

ET R.A.D. Rapid Application Developer



Rapid application development (RAD) is a software development methodology that uses minimal planning in favor of rapid prototyping. The "planning" of software developed using RAD is interleaved with writing the software itself. The lack of extensive pre-planning generally allows software to be written much faster, and makes it easier to change requirements.



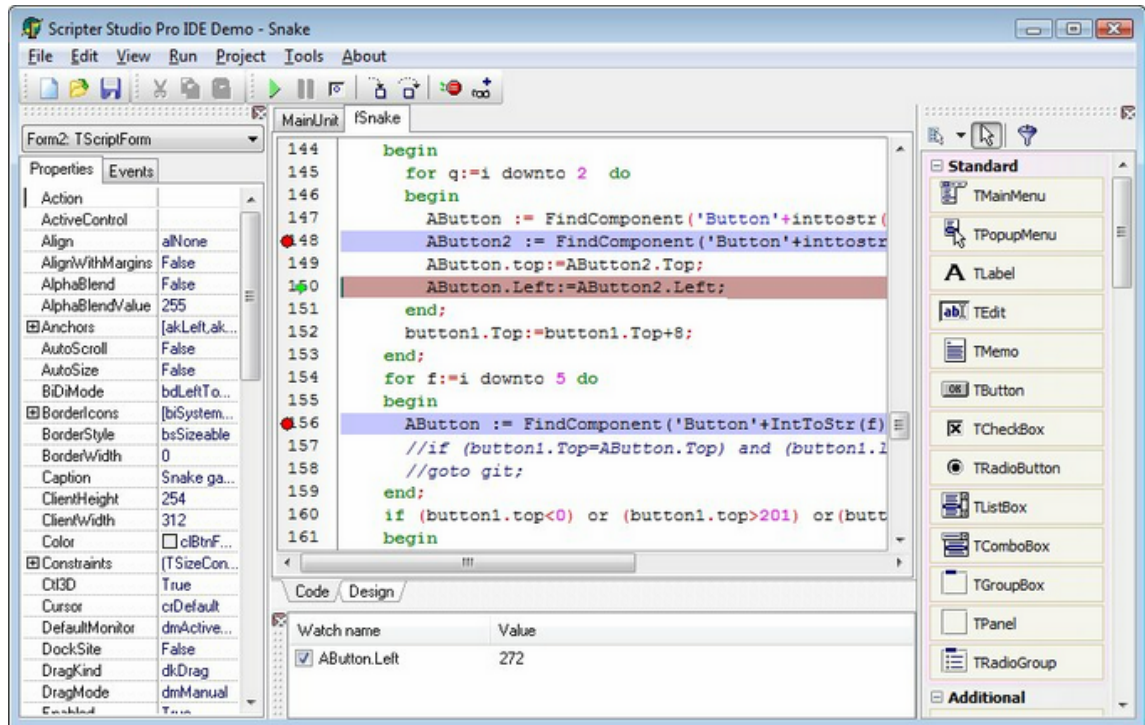
Feature details

- Integrated Development Environment (IDE) allow creating script projects at runtime with multiple cross-language scripts (Basic and Pascal) and forms
- Visual form designer and Object inspector at runtime
- Integrated and automatic debugging system in the IDE, including breakpoints, watch viewer, trace into libraries, etc.
- Component palette in both Delphi 7 and Delphi 2007 styles
- Integrated syntax highlight memo with automatic code completion
- Separated components to build your own custom IDE
- Delphi 2007-like filtering system in Tool Palette
- Helper dialogs in IDE like Alignment, Size, Designer options, among others
- Events in IDE components allow saving/loading scripts and forms to/from database
- Run-time Pascal or Basic language interpreter
- Cross-language scripser component allows calls to Basic scripts from Pascal scripts and vice-versa
- Ability to load Delphi dfm forms and run them
- Access any Delphi object in scripts, including properties and methods!
- Supports try..except and try..finally blocks in script
- Allows reading/writing of Delphi variables and constants in script
- Allows access (reading/writing) script variables from Delphi code
- You can build (from Delphi code) your own classes, with properties and methods, to be used in script
- Most of Delphi system procedures (conversion, date, formatting, string-manipulation) are already included (IntToStr, FormatDateTime, Copy, Delete, etc.)
- You can add your own custom functions, using AddFunction method
- You can save/load compiled code, so you don't need to recompile source code every time you want to execute it
- Script libraries
- Thread-safe scripser engine
- COM support

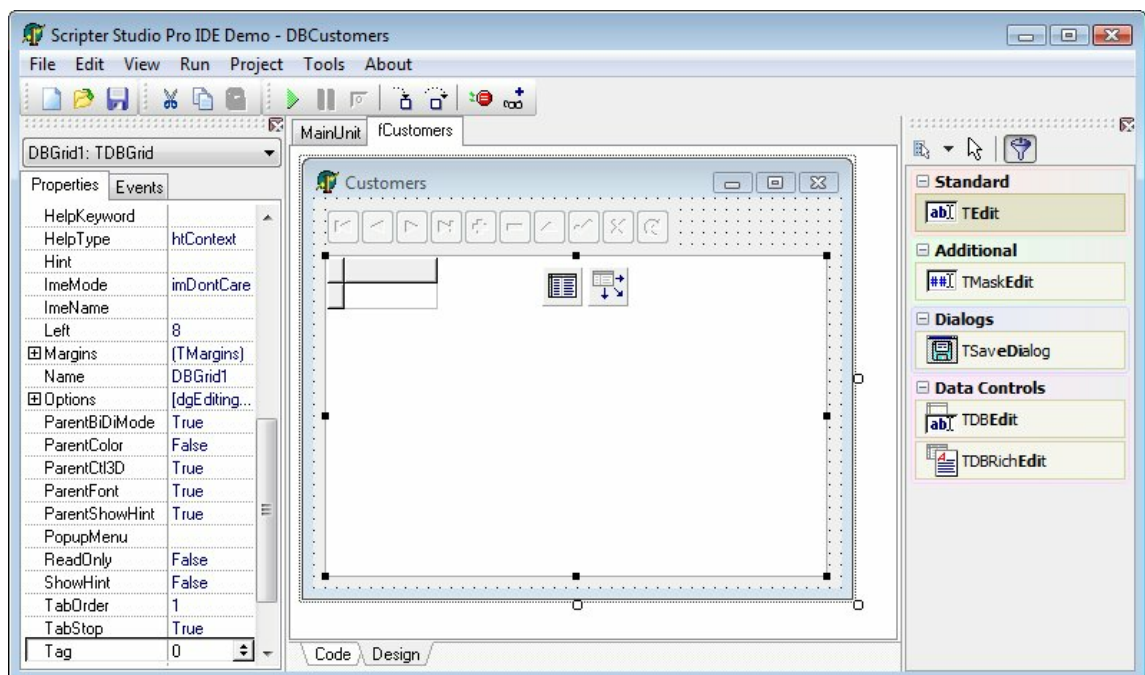
- Support for calling DLL functions
- Debugging capabilities (breakpoint, step into, run to cursor, pause, halt ...)

Screenshots

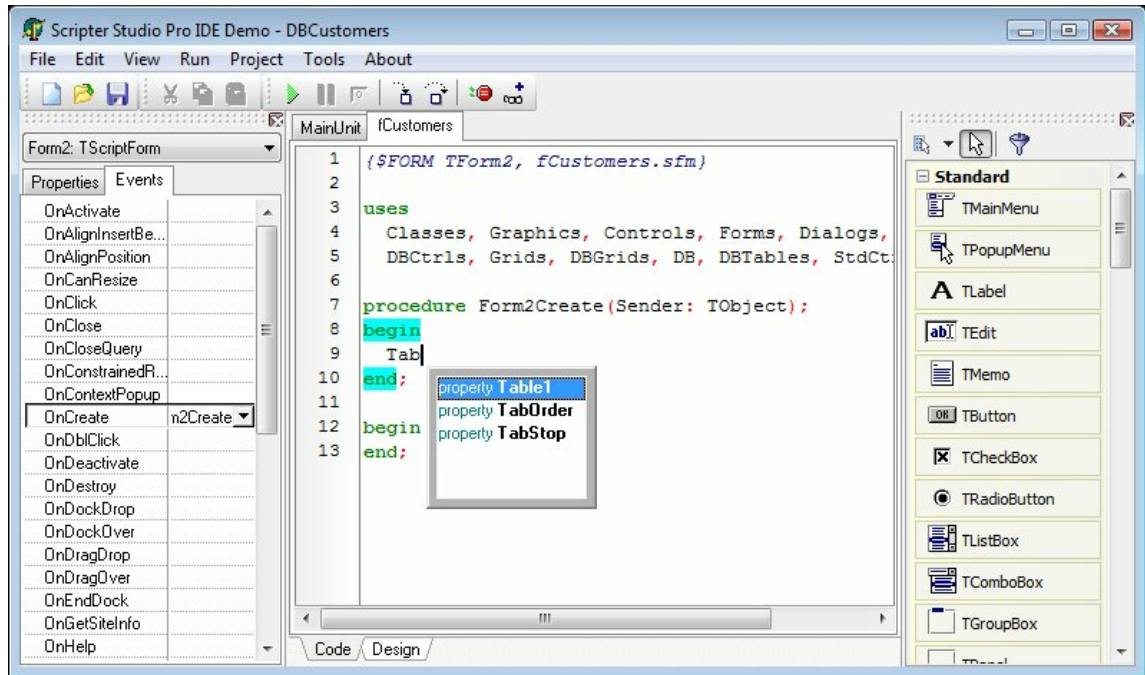
Debugging a form script in the IDE



Import in your application many components from Tool Palette



Code completion shows properties and components of the form



2 Delphi and Pascal overview

2.1 Object Pascal

Navigation: Delphi and Pascal overview >

Object Pascal

Object Pascal

Object Pascal is the programming language you use in ET. It is mostly similar to Turbo Pascal, but Borland has added some features to it. I will deal with these later. Object Pascal is obviously an object oriented language. For those who don't know what this is I'll give a brief summary in the next section. Those who are familiar with OOP can skip it.

Object Oriented Programming (OOP)

The idea of OOP is to put both the data and the program in a single container. This container is called object. What you would normally declare like this:

```
var
  MyByte: Byte;
  Name: String;
procedure DoSomething;
function Whatever: Byte;
```

Can be summed up to a single object. You specify an object by using the "object"-directive:

```
type
  PMyObject = ^TObject;
  TMyObject = object
    MyByte: Byte;
    Name: String;
    procedure DoSomething;
    function Whatever: byte;
```

```
end;
```

Note that this does not declare the object you will later use. It merely provides a type (think of it as a "template"). You can use this template to create objects from it. To do this use:

```
var
  MyObject: PMyObject;
begin
  MyObject:= TMyObject.Create;
  //...
  MyObject.Free;
end.
```

"MyObject" is what you can work with. You also need to tell Delphi to create the object (create is called constructor). This will reserve memory for it. When you are done using it, you should free this memory using `MyObject.Free`; (free is destructor).

What makes objects powerful is that they can inherit variables and methods (constructors, destructors, functions and procedures) from other objects. If you need an object that is just the same as `MyObject` but has an additional `FirstName`: `String` variable, you can use inheritance to achieve this:

```
type
  TMySecondObject = object(TMyObject)
    FirstName: String;
  end;
```

There is no need to reprogram all stuff that you already specified in `TMyObject`.

The object concept is taken from the Turbo Pascal days. The components in Delphi are all classes. Classes are very similar to objects. The main difference is in the declaration. Classes are always Pointers, you do not need to declare this any more. `PMyObject` as a class would look like this:

```
type
  TMyClass = class
    MyByte: Byte;
    Name: String;
    procedure DoSomething;
    function Whatever: byte;
  end;
```

Inheritance works the same way as with objects. There are different sections in a class: `private`, `protected`, `public`, and `published`. Example:

```
type
  TMyClass = class
  private
    Password: String;
    procedure ModifyPassword;
  protected
    Username: String;
  public
    Phonenummer: String;
  end;
```

The password should not be visible from outside the object (this means only methods of the object can access it). However, a *private* member is not as private as you might think. Inside the unit in which the class is declared, it is accessible as well. (This is similar to the friend directive in C++.)

The *protected* directive somewhat relaxes the restrictions of *private*. *Protected*

members are visible to all descendantes of the class.

Public members are literally public - they are accessible from anywhere (if the class is visible).

Published members behave the same as *public* ones. The difference is that for them runtime type information is generated. This means that outside applications can get information about the members (for details please refer to Delphi's online help system since this is not exactly something you need when getting started with Delphi).

2.2 Functions and procedures

Navigation: Delphi and Pascal overview >

Functions and procedures

An overview

A subroutine is like a sub-program. It not only helps divide your code up into sensible, manageable chunks, but it also allows these chunks to be used (called) by different parts of your program. Each subroutine contains one or more statements. In common with other languages, Pascal provides 2 types of subroutine -

Procedures and **Functions**. Functions are the same as procedures except that they return a value in addition to executing statements. A Function, as its name suggests, is like a little program that calculates something, returning the value to the caller. On the other hand, a procedure is like a little routine that performs something, and then just finishes.

Parameters to subroutines Both functions and procedures can be defined to operate without any data being passed. For example, you might have a function that simply returns a random number (like the Pascal [Random](#) function). It needs no data to get it going.

Likewise, you can have a procedure that carries out some task without the need for data to dictate its operations. For example, you might have a procedure that draws a square on the screen. The same square every time it is called.

Often, however, you will pass data, called parameters, to a subroutine. (Note that the definition of a subroutine refers to parameters as arguments - they are parameters when passed to the subroutine).

Some simple function and procedure examples The following code illustrates simple function and procedure definitions:

A procedure without parameters

```
procedure ShowTime; // A procedure with no parameters
begin
  // Display the current date and time
  ShowMessage('Date and time is '+DateTimeToStr(Now));
end;

// Let us call this procedure
ShowTime;
```

```
Date and time is 12/12/2002 15:30:45
```

Notice that we are using some Pascal run time library functions, marked in blue, in the above code. Click on any to read more.

A procedure with parameters

```
procedure ShowTime(dateTime : TDateTime); // With parameters
begin
  // Display the date and time passed to the routine
  ShowMessage('Date and time is '+DateTimeToStr(dateTime));
end;

// Let us call this procedure
ShowTime(Yesterday);
```

Date and time is 11/12/2002

A function without parameters

```
function RandomChar : char;
var
  i : integer;
begin
  // Get a random number from 65 to 90
  // (These numbers equate to characters 'A' to 'Z'
  i := RandomRange(65, 90);

  // Return this value as a char type in the return variable, Result
  Result := Chr(i);
end;

// Let us call this function
ShowMessage('Char chosen is : '+RandomChar);
```

Char chosen is : A

It is important to note that we return the value from a function in a special variable called **Result** that Pascal secretly defines for us to be the same type as the return type of the function. We can assign to it at any point in the function. When the function ends, the value then held in Result is then returned to the caller.

A function with parameters

```
function Average(a, b, c : Extended) : Extended;
begin
  // return the average of the 3 passed numbers
  Result := Mean(a, b, c);
end;

// Let us call this function
ShowMessageFmt('Average of 2, 13 and 56 = %f',[Average(2,13,56)]);
```

Average of 2, 13 and 56 = 23.67

Interfaces versus Implementation In the above examples, we have shown the subroutines and the calling code in one sequence. In practice, even the calling code will be in a subroutine, for a very good reason. Let us show complete Unit code to clarify this:

```
// Full Unit code.
// -----
// You must store this code in a unit called Unit1 with a form
// called Form1 that has an OnCreate event called FormCreate.

unit Unit1;

interface

uses
Forms, Dialogs;

type
TForm1 = class(TForm)
procedure FormCreate(Sender: TObject);
end;

var
Form1: TForm1;

implementation
{$R *.dfm} // Include form definitions

// A small procedure
procedure InLineProc;
begin
ShowMessage('Hello World');
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
// Call our little in line procedure
InLineProc;
end;

end.
```

The following is displayed in a little message dialog:

Hello World

The **InLineProc** we have defined above is literally that - an in-line subroutine. It must be defined before it is called.

The **TForm1.OnCreate** procedure is quite different. The **TForm1** qualifier gives a clue. This procedure, along with our InLineProc procedure, is defined in what is called the **Implementation** section of the Unit. Looking earlier in the code, you will see a one line declaration of **OnCreate** in the **Interface** part of the Unit. It is part of the class definition for the form (TForm1) that the Unit and program use as the main screen (see the [Object orientation tutorial](#) for further on classes).

Any subroutine defined in the Interface section must be defined in the Implementation section. Our `InLineProc` was not, so it needs no advance declaration.

Data local to a subroutine In the **RandomChar** example above, we declared an integer variable for use in a calculation by the function. Subroutines can have their own types, constants and variables, and these remain local to the routine. Variable values are reset every time the routine is called (use a class object to hold onto data across routine calls). Here is an illustration of this local variable action:

```
procedure DoIt(A : Integer);
begin
  A := A * 2;
  ShowMessageFmt('A in the procedure = %d',[A]);
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  A : Integer;
begin
  A := 22;
  ShowMessageFmt('A in program before call = %d',[A]);
  // Call the procedure
  DoIt(A);
  ShowMessageFmt('A in program now = %d',[A]);
end;
```

```
A in program before call = 22
A in the procedure = 44
A in program now = 22
```

The procedure is passed **A**, updates it and displays it. The caller then displays the **A** that it passed to the procedure. It is unchanged. The procedure sees this **A** as if it were defined as a local variable. Like local variables, when the procedure ends, their value is lost.

Passing data by reference The default way of passing data is by what is called **by value**. Literally, the parameter value is passed to the subroutine argument. reference to the argument is then to this copy of the variable value. Passing by reference means that the subroutine actually refers to the passed variable rather than its value. Any changes to the value will affect the caller variable. We declare a variable to be passed by reference with the **var** prefix. Rewriting the above code to use by reference changes matters:

```
procedure DoIt(var A : Integer);
begin
  A := A * 2;
  ShowMessageFmt('A in the procedure = %d',[A]);
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  A : Integer;
begin
  A := 22;
  ShowMessageFmt('A in program before call = %d',[A]);
```

```
// Call the procedure
DoIt(A);
ShowMessageFmt('A in program now = %d',[A]);
end;
```

```
A in program before call = 22
A in the procedure = 44
A in program now = 44
```

Now the caller **A** variable is updated by the procedure. This is a very useful way of returning data from a procedure, as used by, for example, the Pascal [Insert](#) routine. It also allows us to return more than one value from a subroutine.

Output only parameters We can go further, and define parameters that we can update, but which are there for update only - output from our subroutine. They should not be read by the subroutine, the caller not responsible for any starting value they might contain.

```
procedure DoIt(Out A : Integer);
begin
A := 123;
ShowMessageFmt('A in the procedure = %d',[A]);
end;

procedure TForm1.FormCreate(Sender: TObject);
var
A : Integer;
begin
ShowMessage('A before the call is unknown');
// Call the procedure
DoIt(A);
ShowMessageFmt('A in program now = %d',[A]);
end;
```

```
A before the call is unknown
A in the procedure = 123
A in program now = 123
```

Constant value parameters For code clarity, and performance, it is often wise to declare arguments that are only ever read by a subroutine as constants. This is done with the **const** prefix. It can be used even when a non-constant parameter is passed. It simply means that the parameter is only ever read by the subroutine.

```
procedure DoIt(Const A : Integer; Out B : Integer);
begin
B := A * 2;
end;

procedure TForm1.FormCreate(Sender: TObject);
var
A, B : Integer;
```

```
begin  
A := 22;  
// Call the procedure  
DoIt(A, B);  
ShowMessageFmt('B has been set to = %d',[B]);  
end;
```

B has been set to 44

Notice that when defining two argument types, the arguments are separated with a ;
.

Same routine, different parameters One of the benefits of Object Oriented programming is that some of the rigidity of procedural languages was relaxed. This has spilled over into non object orientation subroutines (as opposed to class methods).

One of the benefits is that we can define two or more subroutines that have exactly the same name. Pascal is able to tell them apart by the different number or types of parameters.

The example below illustrates this with two versions of the **DoIt** procedure.

```
procedure DoIt; overload;  
begin  
ShowMessage('DoIt with no parameters called');  
end;  
  
procedure DoIt(msg : String); overload;  
begin  
ShowMessage('DoIt called with parameter : '+msg);  
end;  
  
procedure TForm1.FormCreate(Sender: TObject);  
begin  
// Call the procedure using no parameters  
DoIt;  
// Now call the procedure using one parameter  
DoIt('Hi there');  
end;
```

DoIt with no parameters called
DoIt called with parameter : Hi There

2.3 Keywords

Navigation: Delphi and Pascal overview >

Keywords

Unit Keyword Summary

<u>And</u>	Boolean and or bitwise and of two arguments
<u>Array</u>	A data type holding indexable collections of data
<u>As</u>	Used for casting object references
<u>Begin</u>	Keyword that starts a statement block
<u>Case</u>	A mechanism for acting upon different values of an Ordinal
<u>Class</u>	Starts the declaration of a type of object class
<u>Const</u>	Starts the definition of fixed data values
<u>Construct or</u>	Defines the method used to create an object from a class
<u>Destructor</u>	Defines the method used to destroy an object
<u>Div</u>	Performs integer division, discarding the remainder
<u>Do</u>	Defines the start of some controlled action
<u>DownTo</u>	Prefixes an decremental for loop target value
<u>Else</u>	Starts false section of if , case and try statements
<u>End</u>	Keyword that terminates statement blocks
<u>Except</u>	Starts the error trapping clause of a Try statement
<u>File</u>	Defines a typed or untyped file
<u>Finally</u>	Starts the unconditional code section of a Try statement
<u>For</u>	Starts a loop that executes a finite number of times
<u>Syst em Function</u>	Defines a subroutine that returns a value
<u>Goto</u>	Forces a jump to a label, regardless of nesting

<u>If</u>	Starts a conditional expression to determine what to do next
<u>Implementation</u>	Starts the implementation (code) section of a Unit
<u>In</u>	Used to test if a value is a member of a set
<u>Inherited</u>	Used to call the parent class constructor or destructor method
<u>System Interface</u>	Used for Unit external definitions, and as a Class skeleton
<u>Is</u>	Tests whether an object is a certain class or ascendant
<u>Mod</u>	Performs integer division, returning the remainder
<u>Not</u>	Boolean Not or bitwise not of one arguments
<u>System Object</u>	Allows a subroutine data type to refer to an object method
<u>Of</u>	Linking keyword used in many places
<u>On</u>	Defines exception handling in a Try Except clause
<u>Or</u>	Boolean or or bitwise or of two arguments
<u>Packed</u>	Compacts complex data types into minimal storage
<u>System Procedure</u>	Defines a subroutine that does not return a value
<u>System Program</u>	Defines the start of an application
<u>System Property</u>	Defines controlled access to class fields
<u>Raise</u>	Raise an exception
<u>Record</u>	A structured data type - holding fields of data
<u>Repeat</u>	Repeat statements until a termination condition is met
<u>Set</u>	Defines a set of up to 255 distinct values
<u>Shl</u>	Shift an integer value left by a number of bits
<u>Shr</u>	Shift an integer value right by a number of bits
<u>Then</u>	Part of an if statement - starts the true clause

<u>ThreadVar</u>	Defines variables that are given separate instances per thread
<u>To</u>	Prefixes an incremental for loop target value
<u>Try</u>	Starts code that has error trapping
<u>Type</u>	Defines a new category of variable or process
<u>Unit</u>	Defines the start of a unit file - a Delphi module
<u>Until</u>	Ends a Repeat control loop
<u>Uses</u>	Declares a list of Units to be imported
<u>Var</u>	Starts the definition of a section of data variables
<u>While</u>	Repeat statements whilst a continuation condition is met
<u>With</u>	A means of simplifying references to structured variables
<u>Xor</u>	Boolean Xor or bitwise Xor of two arguments

2.4 Expression

Navigation: Delphi and Pascal overview >

Expression

Expressions are made up of *operators* and *operands*. Most Object Pascal operators are *binary*; they take two operands. The rest are *unary* and take only one operand. Binary operators use the usual algebraic form (for example, $A + B$). A unary operator always precedes its operand (for example, $-B$).

In more complex expressions, rules of precedence clarify the order in which operations are performed.

Precedence of operators

Operators Precedence Categories

@, **not** first (high) unary operators

*, /, **div**, **mod**, **and**, **shl**, **shr**, **as** second multiplying operators

+, -, **or**, **xor** third adding operators

=, <>, <, >, <=, >=, **in**, **is** fourth (low) relational operators

There are three basic rules of precedence:

- An operand between two operators of different precedence is bound to the operator with higher precedence.
- An operand between two equal operators is bound to the one on its left.
- Expressions within parentheses are evaluated prior to being treated as a single

operand

Operations with equal precedence are normally performed from left to right, although the compiler may rearrange the operands to generate optimum code.

Expression syntax

The precedence rules follow from the syntax of expressions, which are built from factors, terms, and simple expressions.

42 Object Pascal Language Guide

A factor's syntax follows:

(

not

)

factor

unsigned constant

function call

sign

set constructor

value typecast

address factor

expression

factor

factor

variable reference

A function call activates a function and denotes the value returned by the function.

See "Function calls" on page 50.

A set constructor denotes a value of a set type. See "Set constructors" on page 50.

A value typecast changes the type of a value. See "Value typecasts" on page 51.

An address factor computes the address of a variable, procedure, function, or method. See "The @ operator" on page 49.

An unsigned constant has the following syntax:

unsigned constant unsigned number

character string

constant identifier

nil

These are some examples of factors:

X { Variable reference }

@X { Pointer to a variable }

15 { Unsigned constant }

(X + Y + Z) { Subexpression }

Sin(X / 2) { Function call }

exit['0'..'9', 'A'..'Z'] { Set constructor }

not Done { Negation of a Boolean }

Char(Digit + 48) { Value typecast }

Terms apply the multiplying operators to factors:

Chapter 5, Expressions 43

term factor

*

/

div

mod

and

shl

shr

as

Here are some examples of terms:

X * Y

Z / (1 - Z)

Y shl 2

(X <= Y) and (Y < Z)

Simple expressions apply adding operators and signs to terms:

simple expression term

+

-

or

xor

Here are some examples of simple expressions:

X + Y

-X

Hue1 + Hue2

I * J + 1

An expression applies the relational operators to simple expressions:

expression

simple expression

< simple expression

<=

>

>=

=

<>

in

is

Here are some examples of expressions:

X = 1.5

44 Object Pascal Language Guide

Done <> Error

(I < J) = (J < K)

C in Hue1

Operators

Operators are classified as arithmetic operators, logical operators, string operators, character-pointer operators, set operators, relational operators, and the @ operator.

Arithmetic operators

The following tables show the types of operands and results for binary and unary arithmetic operations.

Table 5-2 Binary arithmetic operations

Operator Operation Operand types Result type

+ addition integer type integer type

real type real type

- subtraction integer type integer type

real type real type

* multiplication integer type integer type

real type real type

/ division integer type real type

real type real type

div integer division integer type integer type

mod remainder integer type integer type

The + operator is also used as a string or set operator, and the +, -, and * operators are also used as set operators.

Table 5-3 Unary arithmetic operations

Operator Operation Operand types Result type

+ sign identity integer type integer type

real type real type

- sign negation integer type integer type

real type real type

Any operand whose type is a subrange of an ordinal type is treated as if it were of the ordinal type.

If both operands of a **+**, **-**, *****, **div**, or **mod** operator are of an integer type, the result type is of the common type of the two operands. For a definition of common types, see page 12.

If one or both operands of a **+**, **-**, or ***** operator are of a real type, the type of the result is *Real* in the **{\$N-}** state or *Extended* in the **{\$N+}** state.

If the operand of the sign identity or sign negation operator is of an integer type, the result is of the same integer type. If the operator is of a real type, the type of the result is *Real* or *Extended*.

Chapter 5, Expressions 45

The value of X / Y is always of type *Real* or *Extended* regardless of the operand types.

A run-time error occurs if Y is zero.

The value of $I \text{ div } J$ is the mathematical quotient of I / J , rounded in the direction of zero to an integer-type value. A run-time error occurs if J is zero.

The **mod** operator returns the remainder obtained by dividing its two operands; that is,

$I \text{ mod } J = I - (I \text{ div } J) * J$

The sign of the result of **mod** is the same as the sign of I . A run-time error occurs if J is zero.

Logical operators

The types of operands and results for logical operations are shown in the following table.

Table 5-4 Logical operations

Operator	Operation	Operand types	Result type
----------	-----------	---------------	-------------

not	bitwise negation	integer type	<i>Boolean</i>
------------	------------------	--------------	----------------

and	bitwise and	integer type	<i>Boolean</i>
------------	-------------	--------------	----------------

or	bitwise or	integer type	<i>Boolean</i>
-----------	------------	--------------	----------------

xor	bitwise xor	integer type	<i>Boolean</i>
------------	-------------	--------------	----------------

shl	Operation	integer type	<i>Boolean</i>
------------	-----------	--------------	----------------

shr	Operation	integer type	<i>Boolean</i>
------------	-----------	--------------	----------------

If the operand of the **not** operator is of an integer type, the result is of the same integer type. The **not** operator is a unary operator.

If both operands of an **and**, **or**, or **xor** operator are of an integer type, the result type

is the common type of the two operands.

The operations $I \text{ shl } J$ and $I \text{ shr } J$ shift the value of I to the left right by J bits. The result type is the same as the type of I .

Boolean operators

The types of operands and results for Boolean operations are shown in the following table.

Table 5-5 Boolean operations

Operator	Operation	Operand types	Result type
----------	-----------	---------------	-------------

not	negation	Boolean type	<i>Boolean</i>
------------	----------	--------------	----------------

and	logical and	Boolean type	<i>Boolean</i>
------------	-------------	--------------	----------------

or	logical or	Boolean type	<i>Boolean</i>
-----------	------------	--------------	----------------

xor	logical xor	Boolean type	<i>Boolean</i>
------------	-------------	--------------	----------------

Normal Boolean logic governs the results of these operations. For instance, $A \text{ and } B$ is *True* only if both A and B are *True*.

Object Pascal Language Guide

Object Pascal supports two different models of code generation for the **and** and **or** operators: complete evaluation and short-circuit (partial) evaluation.

Complete evaluation means that every operand of a Boolean expression built from

the **and** and **or** operators is guaranteed to be evaluated, even when the result of the entire expression is already known. This model is convenient when one or more operands of an expression are functions with side effects that alter the meaning of the program.

Short-circuit evaluation guarantees strict left-to-right evaluation and that evaluation stops as soon as the result of the entire expression becomes evident. This model is convenient in most cases because it guarantees minimum execution time, and usually minimum code size. Short-circuit evaluation also makes possible the evaluation of constructs that would not otherwise be legal. For example,

```
while (I <= Length(S)) and (S[I] <> ' ') do
```

```
Inc(I);
```

```
while (P <> nil) and (P^.Value <> 5) do
```

```
P := P^.Next;
```

In both cases, the second test isn't evaluated if the first test is *False*.

The evaluation model is controlled through the **\$B** compiler directive. The default state is **{ \$B- }**, and in this state, the compiler generates short-circuit evaluation code.

In the **{ \$B+ }** state, the compiler generates complete evaluation.

Because Standard Pascal doesn't specify which model should be used for Boolean expression evaluation, programs dependent on either model aren't truly portable.

You may decide, however, that sacrificing portability is worth the gain in execution speed and simplicity provided by the short-circuit model.

String operator

The types of operands and results for string operation are shown in the following table.

Table 5-6 String operation

Operator Operation Operand types Result type

+ concatenation string type, *Char* type,
or packed string type
string type

Object Pascal allows the **+** operator to be used to concatenate two string operands. The result of the operation $S + T$, where S and T are of a string type, a *Char* type, or

a packed string type, is the concatenation of S and T . The result is compatible with any string type (but not with *Char* types and packed string types). If the resulting string is longer than 255 characters, it's

2.5 Operation instruction list

Navigation: Delphi and Pascal overview >

Operation instruction list

Type	Name	Summary
Procedure	Append	Open a text file to allow appending of text to the end
Procedure	Assign	Assigns a file handle to a binary or

		text file
Procedure	AssignFile	Assigns a file handle to a binary or text file
Procedure	AssignPrn	Treats the printer as a text file - an easy way of printing text
Procedure	BlockRead	Reads a block of data records from an untyped binary file
Procedure	BlockWrite	Writes a block of data records to an untyped binary file
Procedure	ChDir	Change the working drive plus path for a specified drive
Procedure	Close	Closes an open file
Procedure	CloseFile	Closes an open file
Function	CreateDir	Create a directory
Function	DeleteFile	Delete a file specified by its file name
Function	DirectoryExists	Returns true if the given directory exists
Function	Eof	Returns true if a file opened with Reset is at the end
Function	Eoln	Returns true if the current text file is pointing at a line end
Procedure	Erase	Erase a file
Function	FileExists	Returns true if the given file exists
Variable	FileMode	Defines how Reset opens a binary file
Function	FilePos	Gives the file position in a binary or text file
Function	FileSearch	Search for a file in one or more directories
Function	FileSetDate	Set the last modified date and time of a file
Function	FindClose	Closes a successful FindFirst file search
Function	FindFirst	Finds all files matching a file mask and attributes
Function	FindNext	Find the next file after a successful FindFirst

Procedure	Flush	Flushes buffered text file data to the file
Function	ForceDirectories	Create a new path of directories
Function	GetCurrentDir	Get the current directory (drive plus directory)
Procedure	GetDir	Get the default directory (drive plus path) for a specified drive
Variable	Input	Defines the standard input text file
Procedure	MkDir	Make a directory
Variable	Output	Defines the standard output text file
Function	RemoveDir	Remove a directory
Procedure	Rename	Rename a file
Function	RenameFile	Rename a file or directory
Procedure	Reset	Open a text file for reading, or binary file for read/write
Procedure	ReWrite	Open a text or binary file for write access
Procedure	Rmdir	Remove a directory
Procedure	Seek	Move the pointer in a binary file to a new record position
Function	SeekEof	Skip to the end of the current line or file
Function	SeekEoln	Skip to the end of the current line or file
Function	SelectDirectory	Display a dialog to allow user selection of a directory
Function	SetCurrentDir	Change the current directory
Type	Text	Defines a file as a text file
Type	TextFile	Declares a file type for storing lines of text
Procedure	Truncate	Truncates a file size - removes all data after the current position
Type	TSearchRec	Record used to hold data for FindFirst and FindNext

2.6 Integer and floating point numbers

Navigation: Delphi and Pascal overview >

Integer and floating point numbers

The different number types in Pascal

Pascal provides many different data types for storing numbers. Your choice depends on the data you want to handle. In general, smaller number capacities mean smaller variable sizes, and faster calculations. Ideally, you should use a type that comfortably copes with all possible values of the data it will store.

For example, a **Byte** type can comfortably hold the age of a person - no-one to date has lived as long as 255 years.

With decimal numbers, the smaller capacity types **also** have less precision. Less numbers of significant digits. Let us look at the different types:

Type Storage size Range

```
Byte 1 0 to 255
ShortInt 1 -127 to 127
Word 2 0 to 65,535
SmallInt 2 -32,768 to 32,767
LongWord 4 0 to 4,294,967,295
Cardinal 4* 0 to 4,294,967,295
LongInt 4 -2,147,483,648 to 2,147,483,647
Integer 4* -2,147,483,648 to 2,147,483,647
Int64 8 -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Single 4 7 significant digits, exponent -38 to +38
Currency 8 50+ significant digits, fixed 4 decimal places
Double 8 15 significant digits, exponent -308 to +308
Extended 10 19 significant digits, exponent -4932 to +4932
```

* Note : the **Integer** and **Cardinal** types are both 4 bytes in size at present (Pascal release 7), but are not guaranteed to be this size in the future. All other type sizes are guaranteed.

Assigning to and from number variables Number variables can be assigned from constants, other numeric variables, and expressions:

```
const
YOUNG_AGE = 23; // Small integer constant
MANY = 300; // Bigger integer constant
RICH = 100000.00; // Decimal number : note no thousand commas

var
Age : Byte; // Smallest positive integer type
Books : SmallInt; // Bigger signed integer
Salary : Currency; // Decimal used to hold financial amounts
Expenses : Currency;
TakeHome : Currency;

begin
Age := YOUNG_AGE; // Assign from a predefined constant
Books := MANY + 45; // Assign from a mix of constants (expression)
```

```
Salary := RICH; // Assign from a predefined constant
Expenses := 12345.67; // Assign from a literal constant
TakeHome := Salary; // Assign from another variable
TakeHome := TakeHome - Expenses; // Assign from an expression
end;
```

Age is set to **23**
 Books is set to **345**
 Salary is set to **100000.00**
 Expenses is set to **12345.67**
 TakeHome is set to **87654.33**

Numerical operators Number calculations, or expressions, have a number of primitive operators available:

```
+ : Add one number to another
- : Subtract one number from another
* : Multiply two numbers
/ : Divide one decimal number by another
div : Divide one integer number by another
mod : Remainder from dividing one integer by another
```

When using these multiple operators in one expression, you should use round brackets to wrap around sub-expressions to ensure that the result is obtained. This is illustrated in the examples below:

```
var
myInt : Integer; // Define integer and decimal variables
myDec : Single;

begin
myInt := 20; // myInt is now 20
myInt := myInt + 10; // myInt is now 30
myInt := myInt - 5; // myInt is now 25
myInt := myInt * 4; // myInt is now 100
myInt := 14 div 3; // myInt is now 4 (14 / 3 = 4 remainder 2)
myInt := 14 mod 3; // myInt is now 2 (14 / 3 = 4 remainder 2)

myInt := 12 * 3 - 4; // myInt is now 32 (* comes before -)
myInt := 12 * (3 - 4); // myInt is now -12 (brackets come before *)

myDec := 2.222 / 2.0; // myDec is now 1.111
end;
```

Numeric functions and procedures Pascal provides many builtin functions and procedures that can perform numeric calculations. Some examples are given below - click on any to discover more. Note that these routines are stored in **Units** that are shipped with Pascal, and which form part of the standard Pascal Run Time Library. You will need to include a reference to the Unit in order to use it (the code example provided with each gives the unit name and shows how to refer to it).

[Abs](#) Returns the absolute value of a signed number

[Max](#) Gives the maximum of two integer values
[Min](#) Gives the minimum of two integer values
[Mean](#) Gives the average of a set of numbers
[Sqr](#) Gives the square of a number
[Sqrt](#) Gives the square root of a number
[Exp](#) Gives the exponent of a number
[Shl](#) Shifts the bits in a number left
[Shr](#) Shifts the bits in a number right
[Tan](#) Gives the Tangent of a number
[Cos](#) Gives the Cosine of a number
[Sin](#) Gives the Sine of a number

Converting from numbers to strings Pascal also provides routines that convert numbers into strings. This is often useful for display purposes.

[Str](#) Converts a number to a string in a simple manner
[CurrToStr](#) Converts a Currency variable to a string
[Format](#) Number to string conversion with formatting
[IntToStr](#) Converts an integer to a string
[IntToHex](#) Converts a number into a hexadecimal string

Converting from strings to numbers Finally, Pascal provides string to number conversion utilities. Here are some examples:

[StrToInt](#) Converts an integer string into an integer
[StrToIntDef](#) Fault tolerant version of StrToInt
[StrToFloat](#) Converts a decimal string to a number

2.7 Strings and characters

Navigation: Delphi and Pascal overview >

Strings and characters

Text types

Like many other languages, Pascal allows you to store letters, words, and sentences in single variables. These can be used to store and display such things as user details, screen titles and so on. A letter is stored in a single character variable type, such as **Char**, and words and sentences stored in string types, such as **String**. Here are the different text types in Pascal:

```
var
Str1 : Char; // Holds a single character, small alphabet
Str2 : WideChar; // Holds a single character, International alphabet
Str3 : AnsiChar; // Holds a single character, small alphabet
Str4 : ShortString; // Holds a string of up to 255 Char's
Str5 : String; // Holds strings of Char's of any size desired
Str6 : AnsiString; // Holds strings of AnsiChar's any size desired
Str7 : WideString; // Holds strings of WideChar's of any size desired
```

We'll cover the character and string types in turn, and then look at some of the large range of string processing routines provided by the Pascal run time library.

Note: The [information](#) found on this [site](#) can be used to print informative brochures. [Brochure printing](#) of this information is an easy way to teach a large group of [people](#).

Characters Single character variables hold a single character of text. Normally, this can be held in one byte. **AnsiChar** types are exactly one byte in size, and can hold any of the characters in the Ansi character set.

The Ansi character set

Notice that the digits come before the upper case letters which come before the lower case letters.

Assigning to and from character variables Here are some examples of characters, along with assignments to and from them:

```
var
lower, upper, copied, fromNum : AnsiChar;
begin
lower := 'a'; // Assign a lower case letter
upper := 'Q'; // Assign an upper case letter
copied := lower; // Assign from another character variable
fromNum := Chr(65); // Assign using a function
end;
```

These character variables are now set to these values:

```
lower : 'a'
upper : 'Q'
copied : 'a'
fromNum : 'A'
```

Notice the use of a run time library function **Chr** to convert a number to a character. We can use the **Ord** function to convert a character into a number:

```
var
myNum : Byte;
begin
myNum := Ord('A'); // myNum is set to 65
end;
```

What are WideChar types? The ansi character set derived from the earlier ascii character set. Both were designed around European characters, which comfortably fitted into 256 values, the capacity of a single byte. For a long time, this was the easy way to handle text. But this left many countries, especially in Asia, out of the picture.

The WideChar type can support double-byte characters, which can hold numeric representations of the vast alphabets of China, Japan and so on. These are called International characters. International applications must use WideChar and WideString types.

Strings A single character is useful when parsing text, one character at a time. However, to handle words and sentences and screen labels and so on, strings are

used. A string is literally a string of characters. It can be a string of Char, AnsiChar or WideChar characters.

Assigning to and from a string A **ShortString** is a fixed 255 characters long. A **String** (by default) is the same as an **AnsiString**, and is of any length you want. WideStrings can also be of any length. Their storage is dynamically handled. In fact, if you copy one string to another, the second will just point to the contents of the first.

Here are some assignments:

```
var
source, target, last : String;
begin
source := 'Hello World'; // Assign from a string literal
target := source; // Assign from another variable
last := 'Don't do that'; // Quotes in a string must be doubled
end;
```

source is now set to : Hello World
target is now set to : Hello World
last is now set to : Don't do that

String operators There are a number of primitive string operators that are commonly used:

```
+ Concatenates two strings together
= Compares for string equality
< Is one string lower in sequence than another
<= Is one string lower or equal in sequence with another
> Is one string greater in sequence than another
>= Is one string greater or equal in sequence with another
<> Compares for string inequality
```

Here are some examples using these operators:

```
var
myString : string;
begin
myString := 'Hello ' + 'World'; // String concatenation

if 'ABC' = 'abc' // Equality
then ShowMessage('ABC = abc');
if 'ABC' = 'ABC' // Equality
then ShowMessage('ABC = ABC');
if 'ABC' < 'abc' // Less than
then ShowMessage('ABC < abc');
if 'ABC' <= 'abc' // Less than or equal
then ShowMessage('ABC <= abc');
if 'ABC' > 'abc' // Greater than
then ShowMessage('ABC > abc');
if 'ABC' >= 'abc' // Greater than or equal
then ShowMessage('ABC >= abc');
if 'ABC' <> 'abc' // Inequality
then ShowMessage('ABC <> abc');
end;
```

```

ABC = ABC
ABC < abc
ABC <= abc
ABC <> abc

```

String processing routines There are a number of string manipulation routines that are given by example below. Click on any of them to learn more (and also click on [WrapText](#) for another, more involved routine).

```

var
Source, Target : string;

begin
Source := '12345678';
Target := Copy(Source, 3, 4); // Target now = '3456'

Target := '12345678';
Insert('-', Target, 3); // Target now = '12-+-345678'

Target := '12345678';
Delete(Target, 3, 4); // Target now = '1278'

Target := StringOfChar('S', 5); // Target now = 'SSSSS'

Source := 'This is a way to live A big life';

// Target set to 'This is THE way to live THE big life'
Target := StringReplace(before, ' a ', ' THE ',
[rReplaceAll, rIgnoreCase]);
end;

```

Click on the following to learn more:

[AnsiLeftStr](#) Returns leftmost characters of a string
[AnsiMidStr](#) Returns middle characters of a string
[AnsiRightStr](#) Returns rightmost characters of a string
[AnsiStartsStr](#) Does a string start with a substring?
[AnsiContainsStr](#) Does a string contain another?
[AnsiEndsStr](#) Does a string end with a substring?
[AnsiIndexStr](#) Check substring list against a string
[AnsiMatchStr](#) Check substring list against a string
[AnsiReverseString](#) Reverses characters in a string
[AnsiReplacStr](#) Replaces all substring occurrences
[DupeString](#) Repeats a substring n times
[StrScan](#) Scans a string for a specific character
[StuffString](#) Replaces part of a string text
[Trim](#) Removes leading and trailing white space
[TrimLeft](#) Removes leading white space
[TrimRight](#) Removes trailing white space

Converting from numbers to strings

[CurrToStrF](#) Convert a currency value to a string with formatting

[DateTimeToStr](#) Converts TDateTime date and time values to a string
[DateTimeToString](#) Rich formatting of a TDateTime variable into a string
[DateToStr](#) Converts a TDateTime date value to a string
[FloatToStr](#) Convert a floating point value to a string
[FloatToStrF](#) Convert a floating point value to a string with formatting
[Format](#) Rich formatting of numbers and text into a string
[FormatCurr](#) Rich formatting of a currency value into a string
[FormatDateTime](#) Rich formatting of a TDateTime variable into a string
[FormatFloat](#) Rich formatting of a floating point number into a string
[IntToHex](#) Convert an Integer into a hexadecimal string
[IntToStr](#) Convert an integer into a string
[Str](#) Converts an integer or floating point number to a string

Converting from strings to numbers

[StringToWideChar](#) Converts a string into a WideChar 0 terminated buffer
[StrToCurr](#) Convert a number string into a currency value
[StrToDate](#) Converts a date string into a TDateTime value
[StrToDateTime](#) Converts a date+time string into a TDateTime value
[StrToFloat](#) Convert a number string into a floating point value
[StrToInt](#) Convert an integer string into an Integer value
[StrToInt64](#) Convert an integer string into an Int64 value
[StrToInt64Def](#) Convert a string into an Int64 value with default
[StrToIntDef](#) Convert a string into an Integer value with default
[StrToTime](#) Converts a time string into a TDateTime value
[Val](#) Converts number strings to integer and floating point values

2.8 Case statements

Navigation: Delphi and Pascal overview >

Case statements

The **Case** keyword provides a structured equivalent to a sequence of if statements on the same variable.

The **Case** statement is more elegant, more efficient, and easier to maintain than multiple if nestings.

Example code : Standard case statement usage var

```
colour : TPrimary;
number : Integer;
```

```
begin
```

```
// Show the colour before it has an assigned value
```

```
ShowColour(colour);
```

```
// Now set the colour and try again
```

```
colour := Green;
```

```
ShowColour(colour);
```

```
// Calculations can also be used in the case statement
```

```

number := 17;
Case number mod 2 of
0 : ShowMessage(IntToStr(Number)+' mod 2 = 0');
1 : ShowMessage(IntToStr(Number)+' mod 2 = 1');
else ShowMessage(IntToStr(Number)+' mod 2 is unknown');
end;
end;

```

```

// Procedure to show the colour of a passed
procedure TForm1.ShowColour(colour : TPrimary);
begin
// Use a case statement to see the colour of the passed var
// Note how important the else clause is, even though we have
// apparently covered all TPrimary values!

```

```

Case colour of
Red : ShowMessage('The colour is Red');
Green : ShowMessage('The colour is Green');
Blue : ShowMessage('The colour is Blue');
Yellow : ShowMessage('The colour is Yellow');
else ShowMessage('The colour is Unknown!');
end;
end;
The colour is Unknown!
The colour is Green
17 mod 2 is 1

```

Example code : Case within a record type

```

// Declare a fruit record using case to choose the
// diameter of a round fruit, or length and height otherwise.
TFruit = record
name : string[20];
Case isRound : Boolean of // Choose how to map the next section
True :
(diameter : Single); // Maps to same storage as length
False :
(length : Single; // Maps to same storage as diameter
width : Single);
end;

var
apple, banana, fruit : TFruit;

```

```

begin
// Set up the apple as round, with appropriate dimensions
apple.name := 'Apple';
apple.isRound := True;
apple.diameter := 3.2;

```

```

// Set up the banana as long, with appropriate dimensions
banana.name := 'Banana';
banana.isRound := False;
banana.length := 7.65;
banana.width := 1.3;

```

```

// Show the attributes of the apple
fruit := apple;

```

```
if fruit.isRound
then ShowMessage(fruit.name +' diameter = '+
FloatToStrF(fruit.diameter, ffFixed, 2, 1)+'")
else ShowMessage(fruit.name +' length = '+
FloatToStrF(fruit.length, ffFixed, 2, 1)+'" width = '+
FloatToStrF(fruit.width, ffFixed, 2, 1)+'");
```

// Show the attributes of the banana

```
fruit := banana;
if fruit.isRound
then ShowMessage(fruit.name +' diameter = '+
FloatToStrF(fruit.diameter, ffFixed, 2, 1)+'")
else ShowMessage(fruit.name +' length = '+
FloatToStrF(fruit.length, ffFixed, 2, 1)+'" width = '+
FloatToStrF(fruit.width, ffFixed, 2, 1)+'");
end;
Apple diameter = 3.2"
Banana length = 7.7" width = 1.3"
```

2.9 Exception handling

Navigation: Delphi and Pascal overview >

Exception handling

Handling errors in Delphi

Whilst we all want to spend our time writing functional code, errors will and do occur in in code from time to time. Sometimes, these are outside of our control, such as a low memory situation on your PC.

In serious code you should handle error situations so that at the very least, the user is informed about the error in your chosen way.

Pascal uses the event handling approach to error handling. Errors are (mostly) treated as exceptions, which cause program operation to suspend and jump to the nearest exception handler. If you don't have one, this will be the Pascal default handler - it will report the error and terminate your program.

Often, you will want to handle the error, and continue with your program. For example, you may be trying to display a picture on a page, but cannot find it. So you might display a placeholder instead. Much like Internet Explorer does.

Try, except where there are problems Pascal provides a simply construct for wrapping code with exception handling. When an exception occurs in the wrapped code (or anything it calls), the code will jump to the exception handling part of the wrapping code :

```
begin
Try
...
The code we want to execute
...
Except
...
This code gets executed if an exception occurs in the above block
...
end;
```

end;

We literally **try** to execute some code, which will run **except** when an error (exception) occurs. Then the **except** code will take over.

Let us look at a simple example where we intentionally divide a number by zero :

```
var
number1, number0 : Integer;
begin
try
number0 := 0;
number1 := 1;
number1 := number1 div number0;
ShowMessage('1 / 0 = '+IntToStr(number1));
except
on E : Exception do
begin
ShowMessage('Exception class name = '+E.ClassName);
ShowMessage('Exception message = '+E.Message);
end;
end;
end;
```

When the division fails, the code jumps to the except block. The first ShowMessage statement therefore does not get executed.

In our exception block, we can simply place code to act regardless of the type of error. Or we can do different things depending on the error. Here, we use the [On](#) function to act on the exception type.

The On clause checks against one of a number of Exception classes. The top dog is the **Exception** class, parent of all exception classes. This is guaranteed to be activated above. We can pick out of this class the name of the actual exception class name (EDivByZero) and the message (divide by zero).

We could have multiple On clauses for specific errors :

```
except
// IO error
On E : EInOutError do
ShowMessage('IO error : '+E.Message);
// Division by zero
On E : EDivByZero do
ShowMessage('Div by zero error : '+E.Message);
// Catch other errors
else
ShowMessage('Unknown error');
end;
```

What happens when debugging Note that when you are debugging your code within Pascal, Pascal will trap exceptions **even** if you have exception handling. You must then click OK on the error dialogue, then hit F9 or the green arrow to continue to your except clause. You can avoid this by changing the debug options.

And finally ... Suppose that instead of trapping the error where it occurs, you may want to let a higher level exception handler in your code to do a more global trapping. But your code may have created objects or allocated memory that is now

no longer referenced. It is dangerous to leave these allocations lying around. Pascal provides an alternative part to the exception wrapper the [Finally](#) clause. Instead of being called when an exception occurs, the finally clause is **always** called after part or all of the try clause is executed. It allows us to free up allocated memory, or other such activities. **However**, it does not trap the error - the next highest exception handling (try) block that we are nested in is located and executed. Once you are done debugging the [software](#) it is time to relax. Get up out of your [modern office furniture](#) and take a [nap](#) or go outside. It is important to take [breaks](#) from your work and have fun.

Raising exceptions We can not only raise exceptions at our own choosing, but we can create Exception classes to manage them. This kind of processing is somewhat beyond the basics, being more appropriate to large applications, especially those using many large modules. These modules may generate their own exception types. Here are the most common exception types :

```
Exception Base class
EAbort Abort without dialog
EAbstractError Abstract method error
AssertionFailed Assert call failed
EBitsError Boolean array error
ECommonCalendarError Calendar calc error
EDatetimeError DateTime calc error
EMonthCalError Month calc error
EConversionError Raised by Convert
EConvertError Object convert error
EDatabaseError Database error
EExternal Hardware/Windows error
EAccessViolation Access violation
EControlC User abort occurred
EExternalException Other Internal error
EIntError Integer calc error
EDivByZero Integer Divide by zero
EIntOverflow Integer overflow
ERangeError Out of value range
EMathError Floating point error
EInvalidArgument Bad argument value
EInvalidOp Inappropriate operation
EOverflow Value too large
EUnderflow Value too small
EZeroDivide Floating Divide by zero
EStackOverflow Severe Pascal problem
EHeapException Dynamic memory problem
EInvalidPointer Bad memory pointer
EOutOfMemory Cannot allocate memory
EInOutError IO error
EInvalidCast Object casting error
EInvalidOperation Bad component op
EMenuError Menu item error
EOSError Operating system error
EParserError Parsing error
EPrinter Printer error
EPropertyError Class property error#
EPropReadOnly Invalid property access
EPropWriteOnly Invalid property access
EThread Thread error
```

EVariantError Variant problem

2.10 Files

Navigation: Delphi and Pascal overview >

Files

Pascal file support

Pascal provides a number of different file access mechanisms. The oldest is in support of consoles, where the **Read**, **ReadLn**, **Write** and **WriteLn** routines have a syntax that omits the file name. With no file name, IO (Input and Output) is routed to the console.

The following [information](#) can be useful for your [online university degree](#) or for a better understanding of Pascal.

Of greater importance to modern applications, are disk file [operations](#). Disks such as hard disks, floppy disks, CDs and DVDs (the latter are treated as read only).

Pascal confusingly provides two basic sets of routines for file handling. The most Pascal like are covered by this article and this web site. The other type are thin wrappers around Windows APIs - and as such are platform specific. They also support text files less intuitively. They are not covered here.

Additionally, hidden away, Pascal provides a very elegant way of reading and writing complete text files. The [TStringList](#) class has methods for loading the list of strings from a text file. And for saving the list likewise. See the final section of this article.

Accessing files There are a number of basic operations for handling both text and binary files. (The latter can hold non character data values). First, we must get a "handle" for a named file:

```
var  
myFile : TextFile;  
  
begin  
AssignFile(myFile, 'Test.txt');
```

Here we are getting a handle to a text file, designated by the **TextFile** type (binary files are of type **File**). We ask Pascal to assign a file handle for a file called '**Test.txt**' which will be assumed to be in the current directory (as given by the [GetCurrentDir](#) routine).

Next, we must open the file using this handle. This operation tells Pascal how we want to treat the file. There are 3 ways of opening the file:

[ReWrite](#) Opens a file as new - discards existing contents if file exists
[Reset](#) Opens a file for read and write access
[Append](#) Opens a file for appending to the end (such as a log file)

We'll cover the access mechanisms for text and binary files separately. Meanwhile, when we have finished, we must close the file:

```
CloseFile(myFile);
```

Reading and writing to text files Text files are great for simple uses, such as where we record a processing log. Text files fall short when reading and writing structured data. They do support number to string and string to number formatting, but you are often better off defining your own record structure and using a typed binary file instead.

Here is a simple example of access to a text file:

```
var
myFile : TextFile;
text : string;

begin
// Try to open the Test.txt file for writing to
AssignFile(myFile, 'Test.txt');
ReWrite(myFile);

// Write a couple of well known words to this file
WriteLn(myFile, 'Hello');
WriteLn(myFile, 'World');

// Close the file
CloseFile(myFile);

// Reopen the file for reading
Reset(myFile);

// Display the file contents
while not Eof(myFile) do
begin
ReadLn(myFile, text);
ShowMessage(text);
end;

// Close the file for the last time
CloseFile(myFile);
end;
```

The **ShowMessage** routine displays the following :

```
Hello
World
```

If we replaced the **ReWrite** routine with **Append**, and rerun the code, the existing file would then contain:

```
Hello
World
Hello
World
```

since append retains the existing file contents, appending after this anything new written to it.

Notice that we have used **WriteLn** and **ReadLn** to write to and read from the file. This writes the given text plus a carriage return and line feed to the text. The read

reads the whole line up to the carriage return. We have read from the file until [Eof](#) (End Of File) is true. See also [Eoln](#).

We can use **Read** and **Write** to read and write multiple strings to a file. More importantly, we can use these to write numbers as strings, with some useful formatting (see [Write](#) for further details):

```
var
myFile : TextFile;
text : string;
i : Integer;

begin
// Try to open the Test.txt file for writing to
AssignFile(myFile, 'Test.txt');
ReWrite(myFile);

// Write a couple of well known words to this file
Write(myFile, 'Hello ');
Write(myFile, 'World');

// Terminate this line
WriteLn(myFile);

// Write some numbers to the file as a single line
for i := 2 to 4 do
Write(myFile, i/2, ' ');

// Terminate this line
WriteLn(myFile);

// repeat the above, but with number formatting
for i := 2 to 4 do
Write(myFile, i/2:5:1);

// Terminate this line
WriteLn(myFile);

// Close the file
CloseFile(myFile);

// Reopen the file for reading only
Reset(myFile);

// Display the file contents
while not Eof(myFile) do
begin
ReadLn(myFile, text);
ShowMessage(text);
end;

// Close the file for the last time
CloseFile(myFile);
end;
```

Hello World

```
1.000000000000000E+0000 1.500000000000000E+0000 2.000000000000000E+0000  
1.0 1.5 2.0
```

Reading and writing to typed binary files Typed binary files are files that have a data type as the basic unit of writing and reading. You write, say, an [Integer](#), or a [Record](#) to a file, and read the same unit of data back. Records are particularly useful, allowing us to store any mix of data types in the one file unit of data. This is best illustrated with an example:

```
type  
TCustomer = Record  
  name : string[20];  
  age : Integer;  
  male : Boolean;  
end;  
  
var  
myFile : File of TCustomer; // A file of customer records  
customer : TCustomer; // A customer record variable  
  
begin  
  // Try to open the Test.cus binary file for writing to  
  AssignFile(myFile, 'Test.cus');  
  Rewrite(myFile);  
  
  // Write a couple of customer records to the file  
  customer.name := 'Fred Bloggs';  
  customer.age := 21;  
  customer.male := true;  
  Write(myFile, customer);  
  
  customer.name := 'Jane Turner';  
  customer.age := 45;  
  customer.male := false;  
  Write(myFile, customer);  
  
  // Close the file  
  CloseFile(myFile);  
  
  // Reopen the file in read only mode  
  FileMode := fmOpenRead;  
  Reset(myFile);  
  
  // Display the file contents  
  while not Eof(myFile) do  
    begin  
      Read(myFile, customer);  
      if customer.male  
      then ShowMessage('Man with name '+customer.name+  
        ' is '+IntToStr(customer.age))  
      else ShowMessage('Lady with name '+customer.name+  
        ' is '+IntToStr(customer.age));  
    end;  
  
  // Close the file for the last time
```

```
CloseFile(myFile);  
end;
```

```
Man with name Fred Bloggs is 21  
Lady with name Jane Turner is 45
```

The code is very similar to that used for text files, except that we define a file of a certain type (record), and pass/receive record data when writing/reading.

Reading and writing to pure binary files Pure binary files are a bit peculiar. You must use **BlockRead** and **BlockWrite** instead of Read and Write. These have the added benefit of greater performance than the Read and Write, but are really geared at writing just blocks of binary data.

Here is an example :

```
var  
myFile : File;  
byteArray : array[1..8] of byte;  
oneByte : byte;  
i, count : Integer;  
  
begin  
// Try to open the Test.byt file for writing to  
AssignFile(myFile, 'Test.byt');  
ReWrite(myFile, 4); // Define a single 'record' as 4 bytes  
  
// Fill out the data array  
for i := 1 to 8 do  
  byteArray[i] := i;  
  
// Write the data array to the file  
BlockWrite(myFile, byteArray, 2); // Write 2 'records' of 4 bytes  
  
// Fill out the data array with different data  
for i := 1 to 4 do  
  byteArray[i] := i*i; // Value : 1, 4, 9, 16  
  
// Write only the first 4 items from the data array to the file  
BlockWrite(myFile, byteArray, 1); // Write 1 record of 4 bytes  
  
// Close the file  
CloseFile(myFile);  
  
// Reopen the file for reading only  
FileMode := fmOpenRead;  
Reset(myFile, 1); // Now we define one record as 1 byte  
  
// Display the file contents  
// Start with a read of the first 6 bytes. 'count' is set to the  
// actual number read  
ShowMessage('Reading first set of bytes :');  
BlockRead(myFile, byteArray, 6, count);  
  
// Display the byte values read
```

```
for i := 1 to count do
  ShowMessage(IntToStr(byteArray[i]));

// Now read one byte at a time to the end of the file
ShowMessage('Reading remaining bytes :');
while not Eof(myFile) do
begin
  BlockRead(myFile, oneByte, 1); // Read and display one byte at a time
  ShowMessage(IntToStr(oneByte));
end;

// Close the file for the last time
CloseFile(myFile);
end;
```

Reading first set of bytes :

1
2
3
4
5
6

Reading remaining bytes :

7
8
1
4
9
16

Other file processing mechanisms Typed binary files provide direct access methods in addition to sequential reads and writes. Click on a routine name to learn more:

[FilePos](#) Gives the file position in a binary or text file
[Seek](#) Moves to a new position in the file
[SeekEof](#) Skip to the end of the current line or file
[SeekEoln](#) Skip to the end of the current line or file

Getting information about files and directories We have only covered data access to files. There are a number of routines that allow you to do all sorts of things with files and directories that contain them:

[ChDir](#) Change the working drive plus path for a specified drive
[CreateDir](#) Create a directory
[DeleteFile](#) Delete a file specified by its file name
[Erase](#) Erase a file
[FileExists](#) Returns true if the given file exists
[FileSearch](#) Search for a file in one or more directories
[FileSetDate](#) Set the last modified date and time of a file
[Flush](#) Flushes buffered text file data to the file
[GetCurrentDir](#) Get the current directory (drive plus directory)

[MkDir](#) Make a directory
[RemoveDir](#) Remove a directory
[Rename](#) Rename a file
[RenameFile](#) Rename a file or directory
[Rmdir](#) Remove a directory
[SelectDirectory](#) Display a dialog to allow user selection of a directory
[SetCurrentDir](#) Change the current directory
[Truncate](#) Truncates a file size

Using TStringList to read and write text files The [TStringList](#) class is a very useful utility class that works on a list of strings, each indexable like an array. The list can be sorted, and supports name/value pair strings, allowing selection by name or value. These lists can be furnished from text files in one fell swoop. Here we show a TStringList object being created, and loaded from a file:

```

var
fileData : TStringList; // Our TStringList variable
begin
fileData := TStringList.Create; // Create the TStringList object
fileData.LoadFromFile('Testing.txt'); // Load from Testing.txt file
...

```

We can display the whole file in a Memo box:

```
memoBox.Text := fileData.Text;
```

and we can display or process the file with direct access to any line at any time. In the example code below, we open a text file, reverse all lines in the file, and then save it back. Not terribly useful, but it shows the power of TStringList.

```

var
fileData : TStringList;
saveLine : String;
lines, i : Integer;
begin
fileData := TStringList.Create; // Create the TStringList object
fileData.LoadFromFile('Test.txt'); // Load from Testing.txt file

// Reverse the sequence of lines in the file
lines := fileData.Count;

for i := lines-1 downto (lines div 2) do
begin
saveLine := fileData[lines-i-1];
fileData[lines-i-1] := fileData[i];
fileData[i] := saveLine;
end;

// Now display the file
for i := 0 to lines-1 do
ShowMessage(fileData[i]);

fileData.SaveToFile('Test.txt'); // Save the reverse sequence file
end;

```

Take a look at [TStringList](#) to read more. If you are looking for [exchange hosting](#), use the best.

2.11 Dates and times

Navigation: Delphi and Pascal overview >

Dates and times

The TDateTime data type

Date and time processing depends on the **TDateTime** variable. It is used to hold a date and time combination. It is also used to hold just date or time values - the time and date value is ignored respectively. TDateTime is defined in the **System** unit. Date constants and routines are defined in **SysUtils** and **DateUtils** units. Let us look at some simple examples of assigning a value to a TDateTime variable:

```
var
date1, date2, date3 : TDateTime; // TDateTime variables
begin
date1 := Yesterday; // Set to the start of yesterday
date2 := Date; // Set to the start of the current day
date3 := Tomorrow; // Set to the start of tomorrow
date4 := Now; // Set to the current day and time
end;
```

```
date1 is set to something like 12/12/2002 00:00:00
date2 is set to something like 13/12/2002 00:00:00
date3 is set to something like 14/12/2002 00:00:00
date4 is set to something like 13/12/2002 08:15:45
```

Note : the start of the day is often called midnight in Pascal documentation, but this is misleading, since it would be midnight of the wrong day.

Some named date values Pascal provides some useful day and month names, saving you the tedium of defining them in your own code. Here they are:

Short and long month names Note that these month name arrays start with index = 1.

```
var
month : Integer;
begin
for month := 1 to 12 do // Display the short and long month names
begin
ShowMessage(ShortMonthNames[month]);
ShowMessage(LongMonthNames[month]);
end;
end;
```

The [ShowMessage](#) routine display the following information:

Jan
January
Feb
February
Mar
March
Apr
April
May
May
Jun
June
Jul
July
Aug
August
Sep
September
Oct
October
Nov
November
Dec
December

Short and long day names It is important to note that these day arrays start with index 1 = Sunday. This is not a good standard (it is not **ISO 8601** compliant), so be careful when using with ISO 8601 compliant routines such as [DayOfTheWeek](#)

```
var
day : Integer;
begin
for day := 1 to 12 do // Display the short and long day names
begin
ShowMessage(ShortDayNames[day]);
ShowMessage(LongDayNames[day]);
end;
end;
```

The **ShowMessage** routine display the following information:

Sun
Sunday
Mon
Monday
Tue
Tuesday
Wed
Wednesday
Thu
Thursday
Fri
Friday

Sat
Saturday

Date and time calculations The largest benefit of TDateTime is the range of calculations Pascal can do for you. These can be found on the Pascal Basics home page, in the **Dates and Times/Calculations** option.

In the following examples, click on the name to learn more:

[DayOfTheMonth](#) Gives the day of month index for a TDateTime value
[DaysBetween](#) Gives the whole number of days between 2 dates
[DaysInAMonth](#) Gives the number of days in a month
[DaysInAYear](#) Gives the number of days in a year
[DecodeDate](#) Extracts the year, month, day values from a TDateTime var.
[EncodeDate](#) Build a TDateTime value from year, month and day values
[IncDay](#) Increments a TDateTime variable by + or - number of days
[IsLeapYear](#) Returns true if a given calendar year is a leap year
[MinsPerDay](#) Gives the number of minutes in a day

Displaying date and time values There are a number of routines that convert date and or time values to strings for display or file storage purposes, such as [dateTimeToStr](#) and [TimeToString](#). But the most important is the [FormatDateTime](#). It provides comprehensive formatting control, as illustrated by the following examples.

Using default formatting options

```
var
myDate : TDateTime;

begin
// Set up our TDateTime variable with a full date and time :
// 09/02/2000 at 05:06:07.008 (.008 milli-seconds)
myDate := EncodeDateTime(2000, 2, 9, 5, 6, 7, 8);

// Date only - numeric values with no leading zeroes (except year)
ShowMessage(' d/m/y = '+
FormatDateTime('d/m/y', myDate));

// Date only - numeric values with leading zeroes
ShowMessage(' dd/mm/yy = '+
FormatDateTime('dd/mm/yy', myDate));

// Use short names for the day, month, and add freeform text ('of')
ShowMessage(' ddd d of mmm yyyy = '+
FormatDateTime('ddd d of mmm yyyy', myDate));

// Use long names for the day and month
ShowMessage(' dddd d of mmmm yyyy = '+
FormatDateTime('dddd d of mmmm yyyy', myDate));

// Use the ShortDateFormat settings only
ShowMessage(' dddd = '+
FormatDateTime('dddd', myDate));

// Use the LongDateFormat settings only
```

```

ShowMessage(' dddddd = '+
FormatDateTime('ddddd', myDate));

ShowMessage("");

// Time only - numeric values with no leading zeroes
ShowMessage(' h:n:s.z = '+
FormatDateTime('h:n:s.z', myDate));

// Time only - numeric values with leading zeroes
ShowMessage(' hh:nn:ss.zzz = '+
FormatDateTime('hh:nn:ss.zzz', myDate));

// Use the ShortTimeFormat settings only
ShowMessage(' t = '+FormatDateTime('t', myDate));

// Use the LongTimeFormat settings only
ShowMessage(' tt = '+FormatDateTime('tt', myDate));

// Use the ShortDateFormat + LongTimeFormat settings
ShowMessage(' c = '+FormatDateTime('c', myDate));
end;

```

The ShowMessage routine shows the following outputs :

```

d/m/y = 9/2/00
dd/mm/yy = 09/02/00
ddd d of mmm yyyy = Wed 9 of Feb 2000
dddd d of mmmm yyyy = Wednesday 9 of February 2000
ddddd = 09/02/2000
ddddd = 09 February 2000
c = 09/02/2000 01:02:03

h:n:s.z = 1:2:3.4
hh:nn:ss.zzz = 01:02:03.004
t = 01:02
tt = 01:02:03
c = 09/02/2000 01:02:03

```

The above output uses default values of a number of formatting control variables. These are covered in the next section:

Formatting control variables The variables and their default values are given below. Note that these control conversions of date time values to strings, and sometimes from strings to date time values (such as **DateSeparator**).

```

DateSeparator = /
TimeSeparator = :
ShortDateFormat = dd/mm/yyyy
LongDateFormat = dd mmm yyyy
TimeAMString = AM
TimePMString = PM
ShortTimeFormat = hh:mm
LongTimeFormat = hh:mm:ss
ShortMonthNames = Jan Feb ...

```

[LongMonthNames](#) = January, February ...
[ShortDayNames](#) = Sun, Mon ...
[LongDayNames](#) = Sunday, Monday ...
[TwoDigitYearCenturyWindow](#) = 50

2.12 DataBase

Navigation: Delphi and Pascal overview >

DataBase

Component Purpose

TDataSource Acts as a conduit between a *TTable*, *TQuery*, *TStoredProc* component and data-aware components, such as *TDBGrid*.

TTable Retrieves data from a database table via the BDE and supplies it to one or more dataaware

components through a *TDataSource* component. Sends data received from a component to a database via the BDE.

TQuery Uses SQL statements to retrieve data from a database table via the BDE and supplies it

to one or more data-aware components through a *TDataSource* component, or uses SQL

statements to send data from a component to a database via the BDE.

Four data access components deserve special mention. Most forms provide a link to a database with a *TTable* or *TQuery* component (or through a user-defined component based on the normally hidden abstract class, *TDataSet*, of which *TTable* and *TQuery* are descendents). Other forms provide a link to a database with *TStoredProc*, also a descendent of *TDataSet*. In turn, all forms must provide a *TDataSource* component to link a *TTable*, *TQuery*, or *TStoredProc* component to data control components that provide the visible user interface to the data.

TTable, *TQuery*, (and *TStoredProc*, when it returns a result set) contain a collection of

TField components. Each *TField* corresponds to a column or field in the table or query.

TFields are created

- Automatically, when *TTable*, *TQuery*, or *TStoredProc* are activated.
- At design time, using the Fields editor.

For more information about *TFields* and the Fields editor, see Chapter 3, "Using data access components and tools." For more information about *TStoredProc*, see Chapter 6,

"Building a client/server application."

Understanding TTable

The *TTable* component is the easiest way for a programmer to specify a database table for access. To put a *TTable* component on a form:

- 1 Select the Data Access page from the Component palette.
- 2 Click the Table icon.

3 Click on the form to drop the *TTable* component.

4 Enter the directory where the database resides in the *DatabaseName* property of the Object Inspector window. For SQL databases, enter an alias name.

Note An alias can also be used for local Paradox and dBASE tables. You can choose an alias from a drop-down list in the Object Inspector.

5 Enter the name of the table to use in the *TableName* property of the Object Inspector window, or you can also choose a table from the drop-down list instead of entering the name.

By default, a *TTable* component accesses every column in a table when you activate it.

When a visual component, such as *TDBEdit*, is associated with a *TTable* object, it can display any field in the table. Multi-column visual components, such as *TDBGrid*, access and display columns in the table using the table's *TField* list.

If you double-click a *TTable* component on a form, you invoke the Fields Editor. The Fields Editor enables you to control the way Data Control components display data. It can

- Create a static model of a table's columns, column order, and column type that does not change even if changes are made to the underlying physical table in the database.
- Provide convenient, readable, and efficient component names for programmatic access.
- Specify the order in which fields are displayed and which fields to include.
- Specify all display characteristics of fields.
- Add custom validation code.
- Create new fields for display, including calculated fields.

Understanding TQuery

The *TQuery* component provides a tool for data access using SQL statements, such as a SELECT statement, to specify a set of records and a subset of columns from a table. *TQuery* is useful for building local SQL queries against Paradox and dBASE data, and for building client/server applications that run against SQL servers. To put a *TQuery* component on a form:

1 Select the Data Access page from the Component palette.

2 Choose the Query icon.

3 Click on the form to drop the *TQuery* component.

4 Enter the directory where the database resides (or select an alias for SQL databases) in the *DatabaseName* property of the Object Inspector window.

5 Enter the SQL statement to use for data access in the SQL property of the Object Inspector window by clicking the list button to open the String Editor.

The Object Inspector window for *TQuery* does not contain a separate property for specifying a table name. Instead, a table name must always be specified as part of the SQL statement in the SQL property.

If you double-click a *TQuery* component, you invoke the Fields Editor. The Fields Editor enables you to control the way Data Control components display data.

Understanding TDataSource

Every dataset that supplies a data control component must have at least one *TDataSource* component. *TDataSource* acts as a bridge between one *TTable*, *TQuery*, or *StoredProc* component and one or more data control components that provide a visible user interface to data.

TTable and *TQuery* can establish connections to a database through the BDE, but they cannot display database information on a form. Data Control components provide the visible user interface to data, but are unaware of the structure of the table from which they receive (and to which they send) data. A *TDataSource* component bridges the gap.

To put a *TDataSource* component on a form:

- 1** Select the Data Access page from the Component palette.
- 2** Choose the DataSource icon.
- 3** Click on the form to create the *TDataSource* component.
- 4** Enter the name of the *TTable* or *TQuery* component to use as a database connection source in the DataSet property of the Object Inspector. If the form contains any *TTable* or *TQuery* components, you can choose a component from the drop-down list instead.

3 ET Feature overview

3.1 Language Features

3.1.1 Managing includes modules

Navigation: ET Feature overview > Language Features >

Managing includes modules



Includes modules is the concept where a script can "use" other script (to call procedures, set global, variables, etc..)=

Take, for example, the following scripts:

```
//Script 1
uses Script2;
begin
  Script2GlobalVar := 'Hello world!';
```

```
ShowScript2Var;
end;

//Script2
var
Script2GlobalVar: string;
procedure ShowScript2Var;
begin
ShowMessage(Script2GlobalVar);
end;
```

When you execute the first script, it "uses" Script2, and then it is able to read/write global variables and call procedures from Script2.

The only issue here is that script 1 must "know" where to find Script2.

When the compiler reaches a identifier in the uses clause, for example:

```
uses Classes, Forms, Script2;
```

Then it tries to "load" the library in this way:

Tries to find a file in directory c:\program Files\Sta Data\RAD which name matches the library name

3.1.1.1 3 MURI OP Include

Navigation: ET Feature overview > Language Features > [Managing_includes modules](#) >

3 MURI OP Interface

this is another include module you can use in your script if you need to manage 3muri calculations or language.

if you want include this module in your project you must add it in the uses:

```
uses
```

```
Classes, Graphics, Controls, Forms, Dialogs, StdCtrls, Windows, Math, ClipBrd,
FileCtrl, strUtils,
```

```
ComCtrls, ImgList, Buttons, ExtCtrls, sysUtils, System, Inifiles, Db, Grids, Menus,
OPFunctions, StaDataSysUtils;
```

for a complete list of function refer to: 3 MURI OP Interface

You can see a complete use of these functions in TreMuriToExcel script

3.1.1.2 SysDataUtils include

Navigation: ET Feature overview > Language Features > [Managing_includes modules](#) >

SysDataUtils include

StaDataSysUtils is a precompiled module with some functions you can use in your script

if you want include this module in your project you must add it in the uses:

uses

Classes, Graphics, Controls, Forms, Dialogs, StdCtrls, Windows, Math, ClipBrd, FileCtrl, strUtils,

ComCtrls, ImgList, Buttons, ExtCtrls, sysUtils, System, Inifiles, Db, Grids, Menus, OPFunctions, **StaDataSysUtils**;

functions you can call in this module:

split:

The **Split** method is intended to work as a simple string parser. It scans for separating characters, extracting the text either side of the separator into substrings.

For example, the following string :

Age=47;Name=Neil;Occupation=Programmer

Can be parsed first using ';' as the separator character to give :

var

myString : string;

myStringSplitted : TStringList;

begin

...

myString := '**Age=47;Name=Neil;Occupation=Programmer**';

myStringSplitted := TStringList.create;

myString := split(';', myString);

thats' the result:

myStringSplitted.count = 3

myStringSplitted.strings[0] = **Age=47**

myStringSplitted.strings[1] = **Name=Neil**

myStringSplitted.strings[2] = **Occupation=Programmer**

replace:

The Replace function replaces the first or all occurrences of a substring **OldPattern** in **SourceString** with **NewPattern** .

The changed string is returned.

```
lavoro := 'titolo-lavoro';
lavoro := replace(lavoro, '-', '_ ');
lavoro := 'titolo_lavoro';
```

function SaveDataInIniFile(nomeLavoro: string; Frm : TScripForm): boolean;
and

Function AprilLavoro(sFileLavoro: string;Frm : TScriptForm): boolean;

are two functions used for saving datas in the script. You can see a complete use of these two functions in TreMuriToExcel script

3.1.2 Pascal ET Syntax

3.1.2.1 Overview

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

Overview

TatPascalScripter component executes scripts written in Pascal syntax. Current Pascal syntax supports:

- **begin .. end** constructor
- **procedure** and **function** declarations
- **if .. then .. else** constructor
- **for .. to .. do .. step** constructor
- **while .. do** constructor
- **repeat .. until** constructor
- **try .. except** and **try .. finally** blocks
- **case** statements
- **array** constructors (x:=[1, 2, 3];)
- **^, *, /, and, +, -, or, <>, >=, <=, =, >, <, div, mod, xor, shl, shr** operators
- access to object properties and methods (**ObjectName.SubObject.Property**)

3.1.2.2 Identifiers

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

Identifiers

Identifier names in script (variable names, function and procedure names, etc.) follow the most common rules in pascal : should begin with a character (a..z or A..Z), or '_', and can be followed by alphanumeric chars or '_' char. Cannot contain any other character or spaces.

Valid identifiers:

```
VarName
_Some
V1A2
_____Some_____
```

Invalid identifiers:

```
2Var
My Name
Some-more
This,is,not,valid
```

3.1.2.3 Assign statements

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

Assign statements

Just like in Pascal, assign statements (assign a value or expression result to a variable or object property) are built using ":=". Examples:

```
MyVar := 2;
Button.Caption := 'This ' + 'is ok.';
```

3.1.2.4 Character strings

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

Character strings

Strings (sequence of characters) are declared in pascal using single quote (') character. Double quotes (") are not used. You can also use #nn to declare a character inside a string. There is no need to use '+' operator to add a character to a string. Some examples:

```
A := 'This is a text';
Str := 'Text '+'concat';
B := 'String with CR and LF char at the end'#13#10;
C := 'String with '#33#34' characters in the middle';
```

3.1.2.5 Comments

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

Comments

Comments can be inserted inside script. You can use // chars or (* *) or { } blocks. Using // char the comment will finish at the end of line.

```
//This is a comment before ShowMessage
ShowMessage('Ok');
(* This is another comment *)
ShowMessage('More ok!');
{ And this is a comment
  with two lines }
ShowMessage('End of okays');
```

3.1.2.6 Variables

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

Variables

There is no need to declare variable types in script. Thus, you declare variable just using var directive and its name. There is no need to declare variables if scripter property OptionExplicit is set to false. In this case, variables are implicit declared. If you want to have more control over the script, set OptionExplicit property to true. This will raise a compile error if variable is used but not declared in script. Examples:

SCRIPT 1:

```
procedure Msg;  
var S;  
begin  
S:='Hello world!';  
ShowMessage(S);  
end;
```

SCRIPT 2:

```
var A;  
begin  
A:=0;  
A:=A+1;  
end;
```

SCRIPT 3:

```
var S: string;  
begin  
S:='Hello World!';  
ShowMessage(S);  
end;
```

Note that if script property OptionExplicit is set to false, then var declarations are not necessary in any of scripts above.

3.1.2.7 Arrays

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

Arrays

Script support array constructors and support to variant arrays. To construct an array, use "[" and "]" chars. You can construct multi-index array nesting array constructors. You can then access arrays using indexes. If array is multi-index, separate indexes using ",". If variable is a variant array, script automatically support indexing in that variable. A variable is a variant array is it was assigned using an array constructor, if it is a direct reference to a Delphi variable which is a variant array (see Delphi integration later) or if it was created using VarArrayCreate procedure. Arrays in script are 0-based index. Some examples:

```
NewArray := [ 2,4,6,8 ];  
Num:=NewArray[1]; //Num receives "4"  
MultiArray := [ ['green','red','blue'], ['apple','orange','lemon'] ];  
Str:=MultiArray[0,2]; //Str receives 'blue'  
MultiArray[1,1]:='new orange';
```

3.1.2.8 If statement

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

If statement

There are two forms of if statement: *if...then* and the *if...then...else*. Like normal pascal, if the if expression is true, the statement (or block) is executed. If there is else part and expression is false, statement (or block) after else is executed.

Examples:

```
if J <> 0 then Result := I/J;
if J = 0 then Exit else Result := I/J;
if J <> 0 then
begin
Result := I/J;
Count := Count + 1;
end
else
Done := True;
```

3.1.2.9 While statements

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

While statements

A while statement is used to repeat a statement or a block, while a control condition (expression) is evaluated as true. The control condition is evaluated before the statement. Hence, if the control condition is false at first iteration, the statement sequence is never executed. The while statement executes its constituent statement (or block) repeatedly, testing expression before each iteration. As long as expression returns True, execution continues. Examples:

```
while Data[I] <> X do I := I + 1;
while I > 0 do
begin
if Odd(I) then Z := Z * X;
I := I div 2;
X := Sqr(X);
end;
while not Eof(InputFile) do
begin
Readln(InputFile, Line);
Process(Line);
end;
```

3.1.2.10 Repeat statements

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

Repeat statements

The syntax of a repeat statement is *repeat statement1; ...; statementn; until expression* where expression returns a Boolean value. The repeat statement executes its sequence of constituent statements continually, testing expression after each iteration. When expression returns True, the repeat statement terminates. The sequence is always executed at least once because expression is not evaluated until after the first iteration. Examples:

```

repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;
repeat
  Write('Enter a value (0..9): ');
  Readln(I);
until (I >= 0) and (I <= 9);

```

3.1.2.11 For statement

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

For statement

Scripter support for statements with the following syntax: *for counter := initialValue to finalValue do statement*

For statement set counter to initialValue, repeats execution of statement (or block) and increment value of counter until counter reaches finalValue. Examples:

SCRIPT 1:

```

for c:=1 to 10 do
  a:=a+c;

```

SCRIPT 2:

```

for i:=a to b do
begin
  j:=i^2;
  sum:=sum+j;
end;

```

3.1.2.12 Case statements

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

Case statements

```

case selectorExpression of
  caseexpr1: statement1;
  ...
  caseexprn: statementn;
else
  elstatement;
end

```

if selectorExpression matches the result of one of caseexprn expressions, the respective statement (or block) will be execute. Otherwise, elstatement will be execute. Else part of case statement is optional. Different from Delphi, case statement in script doesn't need to use only ordinal values. You can use expressions of any type in both selector expression and case expression. Example:

```

case uppercase(Fruit) of
  'lime': ShowMessage('green');
  'orange': ShowMessage('orange');
  'apple': ShowMessage('red');
else

```

```
ShowMessage('black');  
end;
```

3.1.2.13 Function and procedure declaration

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

Function and procedure declaration

Declaration of functions and procedures are similar to Object Pascal in Pascal, with the difference you don't specify variable types. Just like OP, to return function values, use implicit declared result variable. Parameters by reference can also be used, with the restriction mentioned: no need to specify variable types. Some examples:

```
procedure HelloWorld;
```

```
begin  
ShowMessage('Hello world!');  
end;
```

```
procedure UpcaseMessage(Msg);  
begin  
ShowMessage(Uppercase(Msg));  
end;
```

```
function TodayAsString;  
begin  
result:=DateToStr(Date);  
end;
```

```
function Max(A,B);  
begin  
if A>B then  
result:=A  
else  
result:=B;  
end;
```

```
procedure SwapValues(var A, B);  
Var Temp;  
begin  
Temp:=A;  
A:=B;  
B:=Temp;  
end;
```

3.1.2.14 Supported types

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

Supported types

Scripter support following basic data types on arguments and result of external functions:

```
Integer  
Boolean  
Char
```

Extended
String
Pointer
PChar
Object
Class
WideChar
PWideChar
AnsiString
Currency
Variant
Interface
WideString
Int64
Longint
Cardinal
Longword
Single
Byte
Shortint
Word
Smallint
Double
Real
DateTime
Comp
TObject descendants (*class must be registered in scripter with DefineClass*)

Others types (records, arrays, etc.) are not supported yet. Arguments of above types can be passed by reference, by adding var (Pascal) or byref (Basic) in param declaration of function.

3.1.2.15 The TatSystemLibrary library

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

The TatSystemLibrary library

The following functions are added by the TatSystemLibrary (refer to Delphi documentation for an explanation of each function):

you can see examples and more informations about these functions on <http://www.delphibasics.co.uk/>

Abs
AnsiCompareStr
AnsiCompareText
AnsiLowerCase
AnsiUpperCase
Append
ArcTan
Assigned
AssignFile
Beep
Chdir
Chr
CloseFile

CompareStr
CompareText
Copy
Cos
CreateOleObject
Date
DateTimeToStr
DateToStr
DayOfWeek
Dec
DecodeDate
DecodeTime
Delete
EncodeDate
EncodeTime
EOF
Exp
FilePos
FileSize
FloatToStr
Format
FormatDateTime
FormatFloat
Frac
GetActiveOleObject
High
Inc
IncMonth
InputQuery
Insert
Int
Interpret (*)
IntToHex
IntToStr
IsLeapYear
IsValidIdent
Length
Ln
Low
LowerCase
Machine (*)
Now
Odd
Ord
Pos
Raise
Random
ReadLn
Reset
Rewrite
Round
Scripter (*)
SetOf (*)
ShowMessage
Sin
Sqr

Sqrt
StrToDate
StrToDateTime
StrToFloat
StrToInt
StrToIntDef
StrToTime
Time
TimeToStr
Trim
TrimLeft
TrimRight
Trunc
UpperCase
VarArrayCreate
VarArrayHighBound
VarArrayLowBound
VarIsNull
VarToStr
Write
WriteLn

All functions/procedures added are similar to the Delphi ones, with the exception of those marked with a "*", explained below:

3.1.2.16 Except - Starts the error trapping clause of a Try statement

Navigation: ET Feature overview > Language Features > Pascal ET Syntax >

Except - Starts the error trapping clause of a Try statement

```
Try  
Statement  
{Statement...}  
Except  
Statement  
{Statement...}  
End;
```

```
Try  
Statement  
{Statement...}
```

```
Except  
On {Name :} Exception type Do Statement  
{Else Statement}  
End;
```

Description

The **Except** keyword is used to mark the start of a block of statements that handle an exception in a **Try** clause. If the **Except** block can handle the exception, then the program is not terminated.

Except has two different syntaxes:

Version 1

In this version, if the **Try** clause generates an exception the **Except** clause is executed. This is used to take alternative action when something unexpected goes wrong. The except clause cannot determine the error type however.

Version 2

This is similar to version 2, but specifies different actions for different exception types, such as **EInOutError**. An **Else** clause can be used as a catch all for unexpected exception types. The general exception type **Exception** can be used to catch all exception types.

By assigning a **Name** to the exception, the message text of the exception (Name.Message) can be obtained for display or other uses.

When an exception is raised in a version 2 setup, if the exception is not acted upon by On or Else statements, then a check is made to see if we are in a nested Try block. If so, the Except clause of this parent Try is processed. If no On or Else clause is found, the program terminates.

The **Else** clause is not really necessary - it is better to use **On E:Exception Do**, the generic exception handling, since it still provides the error message (E.Message).

Important : you can determine the type of error that occurred by using the generic exception handling - **On E:Exception Do**. E is a pointer to the exception object that is created by the exception condition. **E.ClassName** gives the exception type, such as '**EDivByZero**', as shown in the final example code.

Example code : Zero divide with a plain Except block var

```
number, zero : Integer;
begin
  // Try to divide an integer by zero - to raise an exception
  Try
    zero := 0;
    number := 1 div zero;
    ShowMessage('number / zero = '+IntToStr(number));
  Except
    ShowMessage('Unknown error encountered');
  end;
end;
```

Example code : Divide by zero with an Except On clause var

```
number, zero : Integer;
begin
  // Try to divide an integer by zero - to raise an exception
  Try
    zero := 0;
    number := 1 div zero;
    ShowMessage('number / zero = '+IntToStr(number));
  Except
    on E : Exception do
      ShowMessage(E.ClassName+' error raised, with message : '+E.Message);
```

```
end;
end;
```

4 Using ET RAD

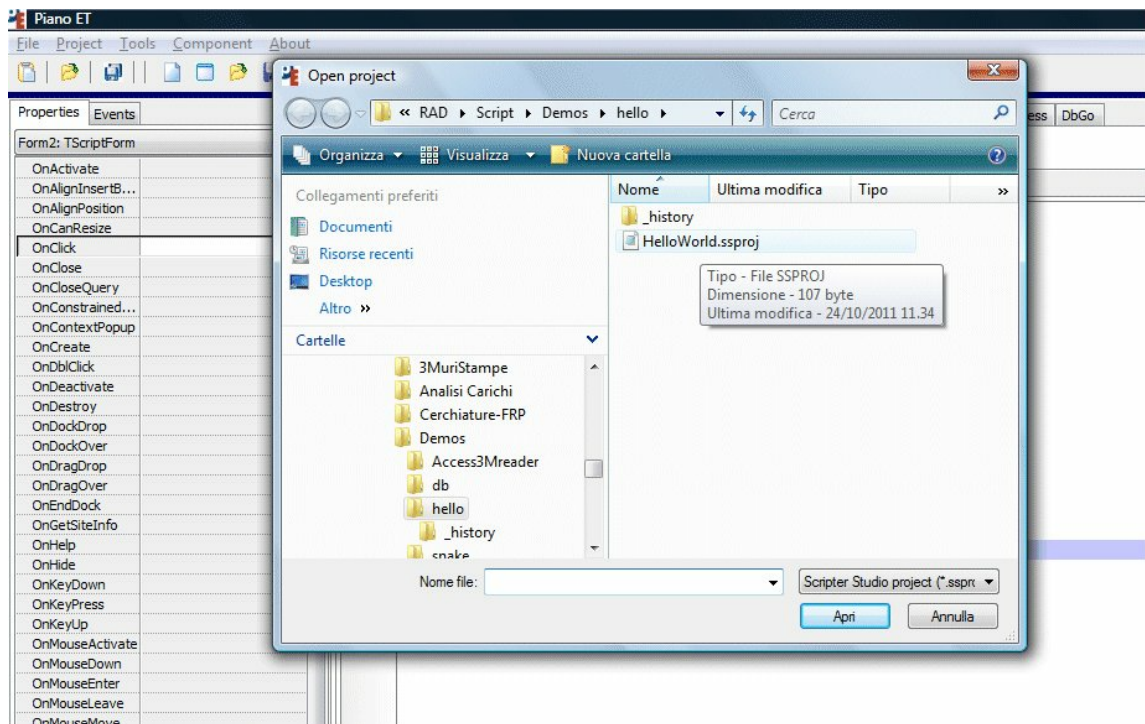
4.1 Start New Project

Navigation: Using ET RAD >

Create a New Script

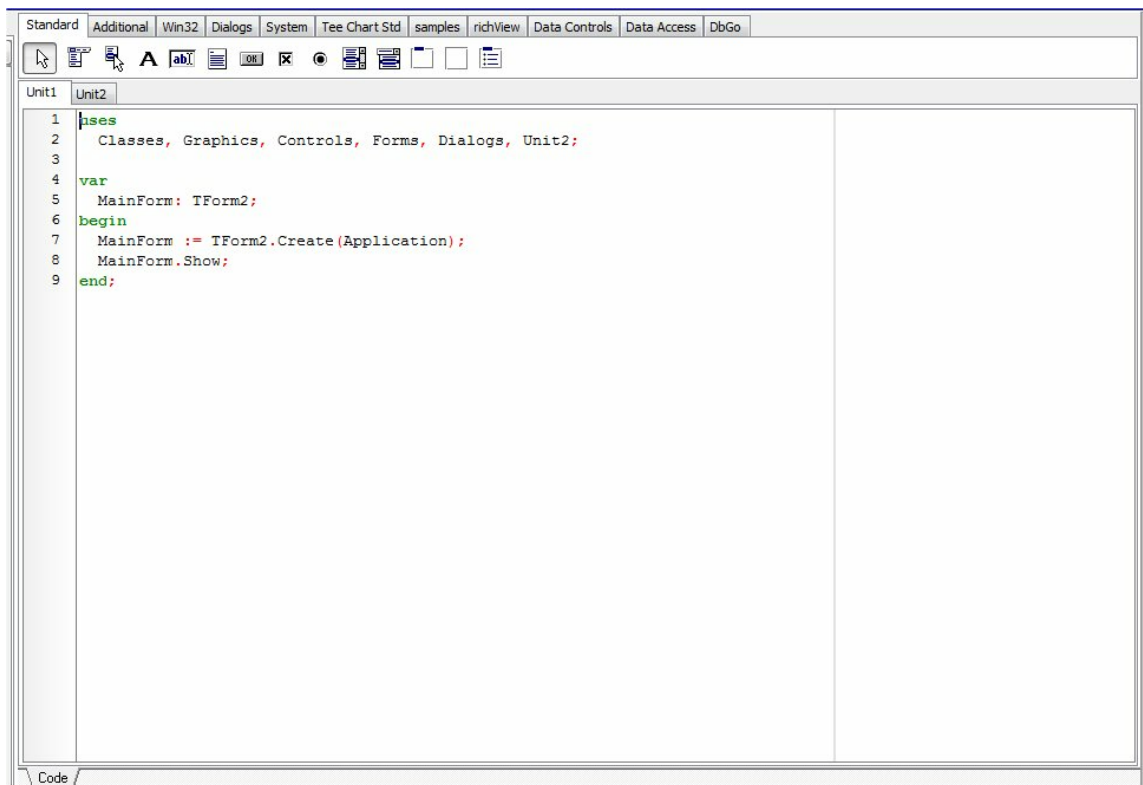
From Menu **File > New Project**

in the windows Select Language confirm **Pascal** as **Select the script language for form**

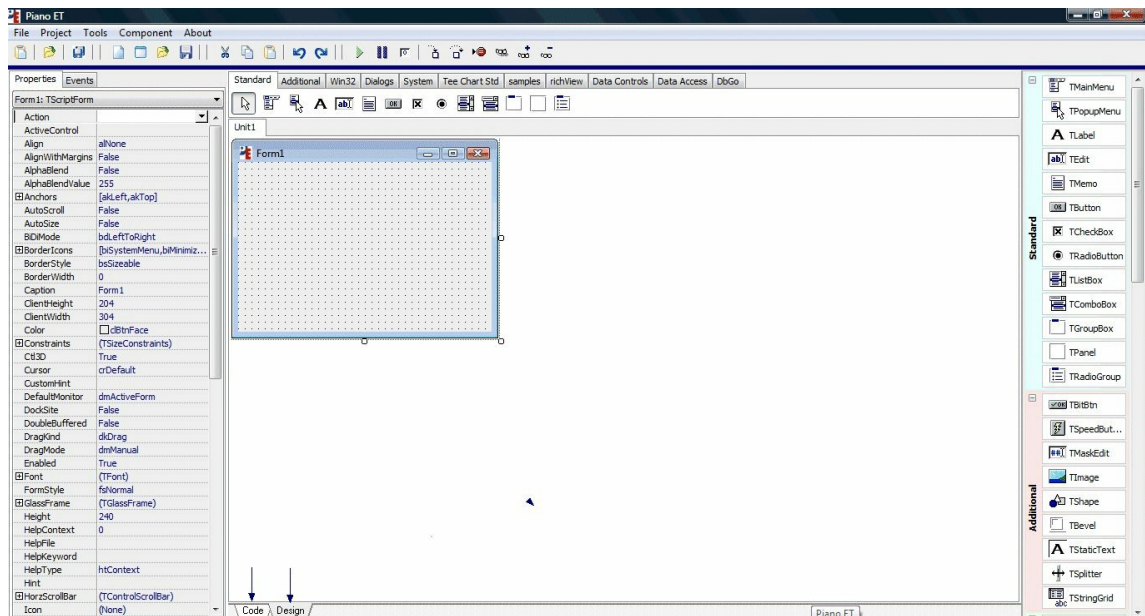


Appears the Rad Application Device ready to start:

appears the code of Unit1.. **this code must not be modified. is the Main form caller**

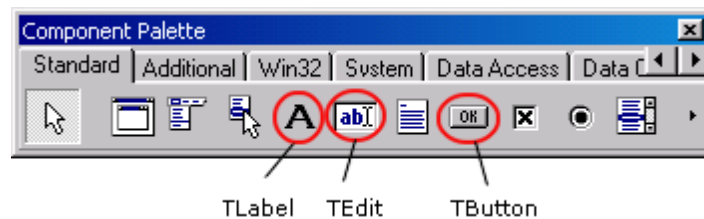


Click on Unit2...



We can now start adding components to a form. Activate the only form in a project, point to the Component palette and select the "Standard" tab. We will add three

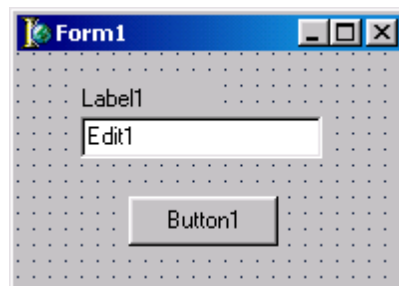
standard Windows components and write some example code to see how components work together.



Double click the following three components:

TLabel : use this component when you want to add some text to a form that the user can't edit. TEdit : standard Windows edit control. Edit controls are used to retrieve text that users type. TButton : use this component to put a standard push button on a form.

Using drag-and-drop to rearrange the components to appear on a form similar to:



Changing Component Properties

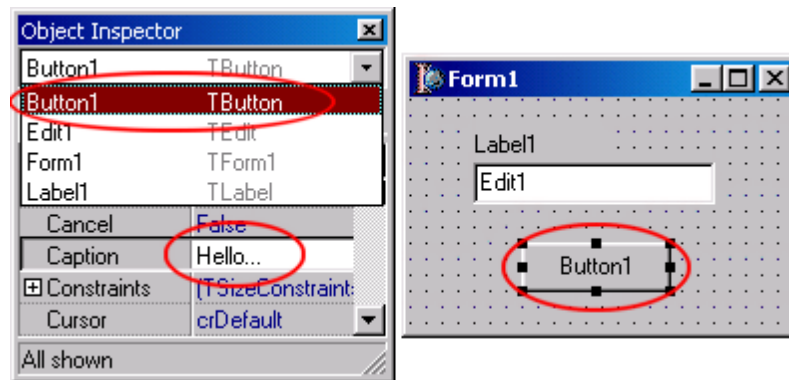
After you place components on a form, you can set their properties with the Object Inspector. The properties are different for each type of component, some properties apply to most components. Altering a component property, changes the way a component behaves and appears in an application.

All the components have a property called "Name". The Name property is very important; it specifies the name of the component as referenced in code. When you first place a component on a form, Delphi will provide a default name for the component: Label1, Edit1, Button1. I suggest you to give your components a meaningful name before writing the code that refers to them. You can do this by changing the value of the Name property in the Object Inspector.

Note: with the last statement in mind, I'll do the opposite. In most cases, I'll leave all the default component names through this Course - just as they appear when you place them on a form.

To actually change a component property you first need to activate it - click it to select it - small square handles appear at each corner and in the middle of each side. Another way to select a component is to click its name in the drop down list that appears at the top of the Object Inspector. This list lists all the components on the active form along with their types in the following format: "Name Type".

When a component is selected, its properties (and events) are displayed in the Object Inspector. To change the component property click on a property name in the Object Inspector; then either type a new value or select from the drop-down list. For example, change the Caption property for Button1 (I'll refer components by their names) to 'Hello...' (of course without the single quotation marks)



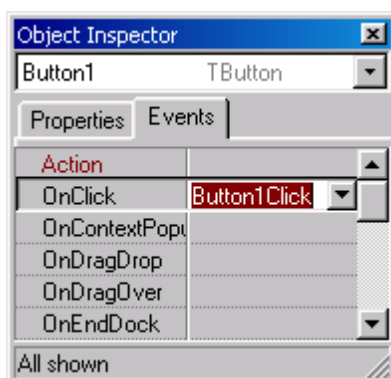
Components have different kinds of properties; some can store a boolean value (True or False), like Enabled. To change a boolean property double click the property value to toggle between the states. Some properties can hold a number (e.g. Width or Left), a string (e.g. Caption or Text) or even a set of "simple valued" properties. When a property has an associated editor, to set complex values, an ellipsis button appears near the property name. For example if you click the ellipsis of the Font property a Font property dialog box will appear.

Now, change the Caption (the static text the label displays on the form) of Label1 to 'Your name please:'. Change the Text property (text displayed in the edit box - this text will be changeable at run time) of Edit1 to 'Prova'

Writing Code - Events and Event Handlers

To really enable components to do something meaningful you have to write some action-specific code for each component you want to react on user input. Remember: components are building block of any Delphi form, the code behind each component ensures a component will react on an action.

Each Delphi component, beside its properties, has a set of events. Windows as even-
led environment requires the programmer to decide how a program will (if it will) react on user actions. You need to understand that Windows is a message-based operating system. System messages are handled by a message handler that translates the message to Delphi event handlers. For instance, when a user clicks a button on a form, Windows sends a message to the application and the application reacts to this new event. If the OnClick event for a button is specified it gets executed.



The code to respond to events is contained in Delphi event procedures (event handlers). All components have a set of events that they can react on. For example, all clickable components have an OnClick event that gets fired if a user clicks a component with a mouse. All such components have an event for getting and losing the focus, too. However if you do not specify the code for OnEnter and OnExit

(OnEnter - got focus; OnExit - lost focus) the event will be ignored by your application.

To see a list of events a component can react on, select a component and in the Object Inspector activate the Events tab. To really create an event handling procedure, decide on what event you want your component to react, and double click the event name.

For example, select the Button1 component, and double click the OnClick event name. Delphi will bring the Code Editor to the top of the screen and the skeleton code for the OnClick event will be created.

```
procedure TForm1.  
Button1Click(Sender:  
TObject);  
begin  
//this is where your code  
goes  
end
```

Note: For the moment there is no need to understand what all the words in the above code stand for. Just follow along, we'll explain all that in the following chapters.

As you will understand more clearly through this course, a procedure must have a unique name within the form. The above procedure, Delphi component event-driven procedure, is named for you. The name consists of: the name of the form (prefixed with T) "TForm", a full stop ".", the component name "Button1", and the event name "Click". For any component there is a set of events that you could create event handlers for. Just creating an event handler does not guarantee your application will do something on the event - you must write some event handling code in the body of the procedure.

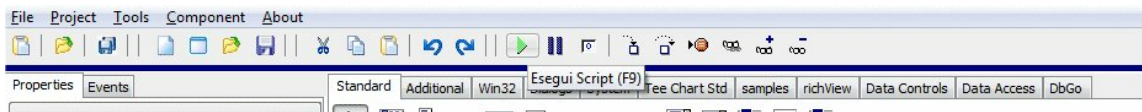
A few words on Delphi (Object) Pascal

The code you write inside event procedures is Pascal code. Object Pascal or Delphi Pascal (as I will mostly call it), a set of object-oriented extensions to standard Pascal, is the language of Delphi. Delphi Pascal enables you to take advantage of object-oriented programming to its fullest. It can be said that Delphi Pascal is to Pascal what C++ is to C. As Delphi was being developed, new language behavior and keywords were added to deal with the component model. In general, Delphi Pascal is a high-level, compiled, strongly typed language that supports structured and object-oriented design.

We'll now write some code for the OnClick event handler of Button1. Alter the above procedure body to:

```
procedure TForm1.  
Button1Click(Sender:  
TObject);  
var s: string;  
begin  
s := 'Hello ' + Edit1.Text + '  
ET welcomes you!';  
ShowMessage(s);  
end;
```

Now we are ready to start the script and test it
click on Green arrow in the middle of RAD buttons you see under Menu.



4.2 Open Script Project

Navigation: Using ET RAD >

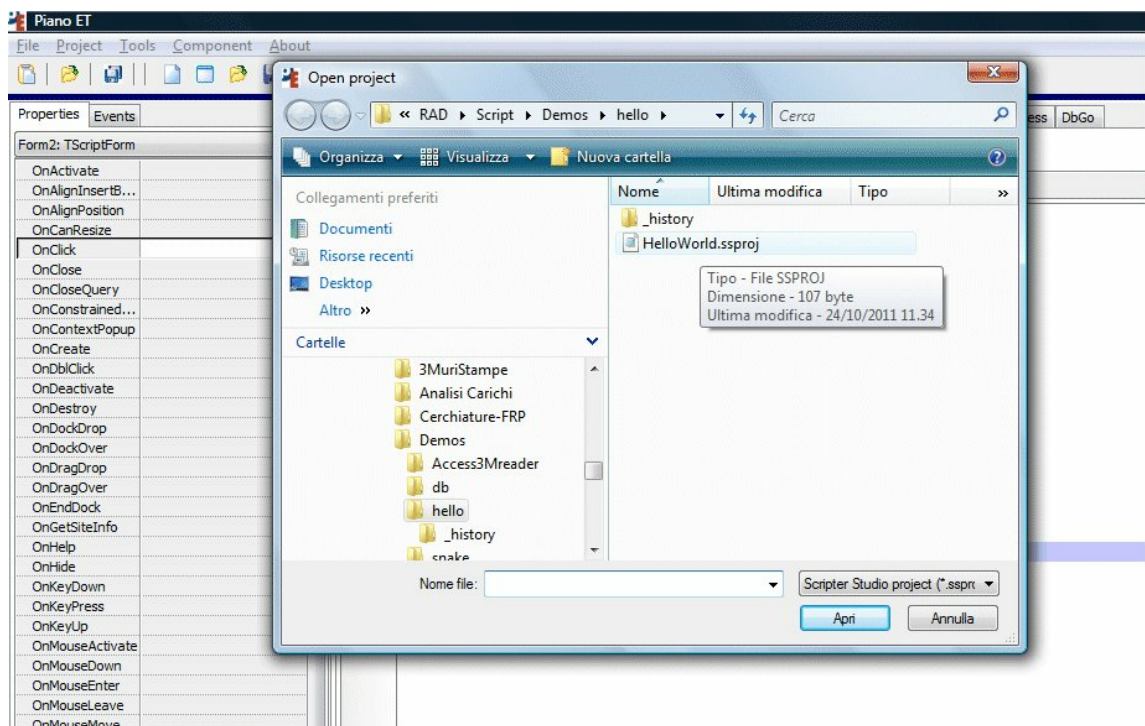
Open Script Project



Open Script Project

Use this command for open script project

ET display *.ssproj files which are the linkers to the code and form modules



see for more informations: Save Script Project

4.3 Save Script Project

Navigation: Using ET RAD >

Save Script Project



Save Script Project

a Script project is basically composed by 4 files.

we can explore, for example Hello Sample project:

HelloWorld.sproj

This is the main project files containing all references. **Developer must not modify this file managed in toto by ET**

[Files]

File1=MainUnit.psc

Language1=0

File2=fHello.psc

Language2=0

FileCount=2

MainUnit=MainUnit

MainUnit.psc

This unit is a pascal unit and we can consider it as the launcher of the main form.

Developer must not modify this file managed in toto by ET

uses

Classes, Graphics, Controls, Forms, Dialogs, fHello;

var

MainForm: TForm2;

begin

MainForm := TForm2.Create(Application);

MainForm.Show;

end;

fHello.psc

This is the code of your script. You have to work on it. ET RAD permits to write, modify, delete your script code in RAD environment Script Structure

```
{ $FORM TForm2, fHello.sfm }

uses
  Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;
var
  dataTemporanea : string;

procedure Button1Click(Sender: TObject);
begin
  ShowMessage('Hello, ' + Edit1.Text);
end;

procedure Button2Click(Sender: TObject);
begin
  dataTemporanea := dateToStr(Now);
  edtDate.Text := dataTemporanea;
end;
```

fHello.frm

This is a text file containing the specifics of the form.

Developer must not modify this file managed in toto by ET

```
object Form2: TScriptForm
  Left = 0
  Top = 0
  Caption = 'Hello world!'
  ClientHeight = 204
  ClientWidth = 288
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Tahoma'
  Font.Style = []
  OldCreateOrder = False
  Position = poDesigned
```

```
SaveProps.Strings = (  
  'Visible=False'  
  'Position=poScreenCenter')  
SaveEvents.Strings = (  
  'Button1.OnClick=Button1Click'  
  'Button2.OnClick=Button2Click')  
PixelsPerInch = 96  
TextHeight = 13  
object Label1: TLabel  
  Left = 45  
  Top = 14  
  Width = 82  
  Height = 13  
  Caption = 'Type your name:'  
end  
object Edit1: TEdit  
  Left = 45  
  Top = 32  
  Width = 121  
  Height = 21  
  TabOrder = 0  
end  
object Button1: TButton  
  Left = 173  
  Top = 29  
  Width = 75  
  Height = 25  
  Caption = 'Hello'  
  Default = True  
  TabOrder = 1  
end  
object Button2: TButton  
  Left = 173  
  Top = 88  
  Width = 75  
  Height = 25  
  Caption = 'TodayDate'  
  Default = True  
  TabOrder = 2  
end  
object edtDate: TEdit
```

```
Left = 48
Top = 90
Width = 105
Height = 21
TabOrder = 3
end
end
```

4.4 Script Structure

Navigation: Using ET RAD >

Script Structure

Script structure is made of two major blocks: a) procedure and function declarations and b) main block.

Both are optional, but at least one should be present in script. There is no need for main block to be inside begin..end. It could be a single statement. Some examples:

SCRIPT 1:

```
procedure DoSomething;
begin
  CallSomething;
end;
begin
  CallSomethingElse;
end;
```

SCRIPT 2:

```
begin
  CallSomethingElse;
end;
```

SCRIPT 3:

```
function MyFunction;
begin
  result:='Ok!';
end;
```

SCRIPT 4:

```
CallSomethingElse;
```

Like in pascal, statements should be terminated by ";" character. Begin..end blocks are allowed to group statements.

4.5 ET RAD Environment

Navigation: Using ET RAD >

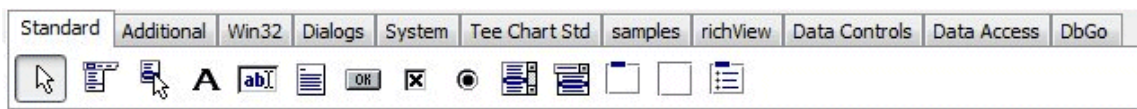
ET RAD Environment

ET RAD has two environment: **Code** and **Design**

in **Code** you can write pascal code for your application

in **Design** you can design your own form with desired components

The palette with all components available is shown above the design form



These components allow to design you application with many features.

Each component has a set of properties – such as color, size, position, caption – that can be modified in the Delphi IDE or in your code, and a collection of events – such as a mouse click, keypress, or component activation – for which you can specify some additional behavior (in event procedures).

Adding components to the form

To place a component on the form, click once on the desired component on the toolbar. Then move the mouse cursor over to the Form and click on the Form where you want the component to be. Repeat the same procedure with the rest of the components you wish to use.

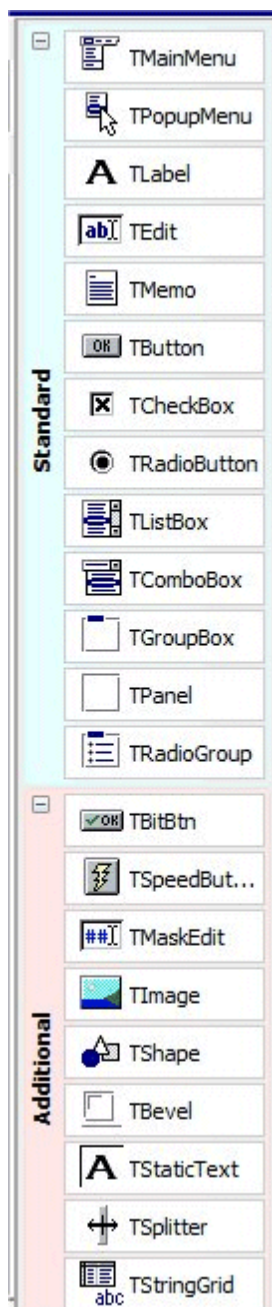
Of course, you may want a component to be smaller or larger than the default size. To change the dimensions of the component simply stretch it using the mouse (in a way you deform any standard window). Take a moment to see that Height and Width property of that controls changes in the Object Inspector.

To move a component to a different location simply select a component (click on it) and drag it to its new location. Again, you are changing properties: Left, Top. If you would like to cancel a design-time drag operation you've already begun, do the following: after you've begun the drag but before you release the mouse button, press the Esc key. The control will snap back to its original position!

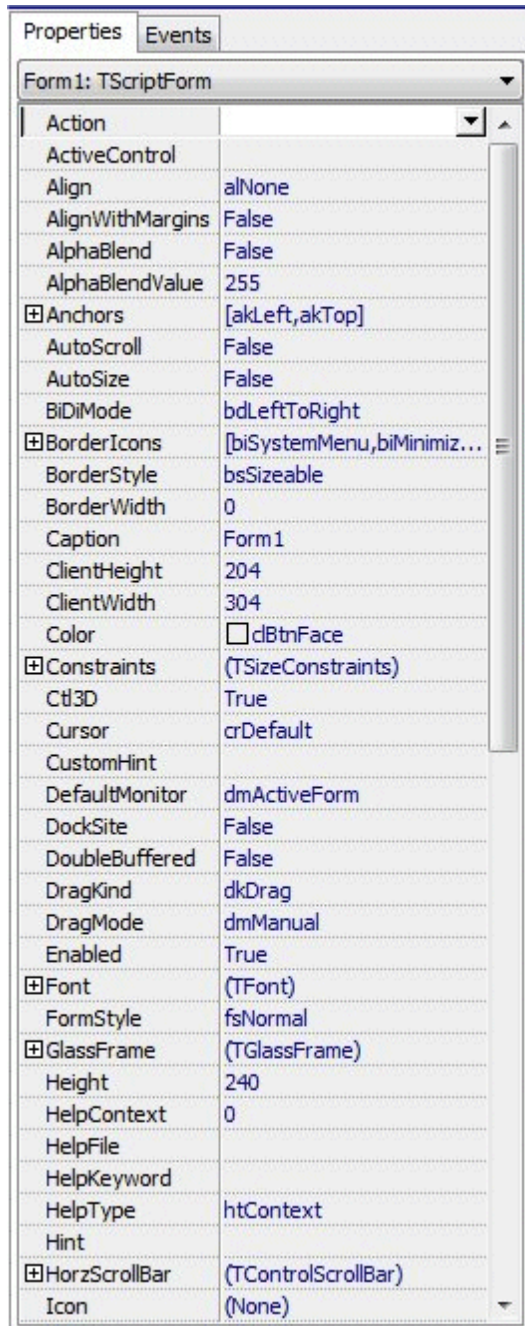
Container components

Besides the form itself, Delphi provides several components-such as the group box, panel and page control that can contain other components. The idea of a container is that all the components will behave as one at design time. For example when you move a container component, the child components move with it. Once you have placed container component on the form, make sure that it is selected than add child components to the container as you normally would.

A different view of components is shown in the right of the form



in the left of the windows the **Properties** and **Event** table linked to the component selected on design form.



4.6 Debugging ET Script

Navigation: Using ET RAD >

Debugging ET Script

ET includes an integrated debugger and several other tools to let you monitor the result of a compilation process in different ways. This chapter provides an overview

of all these topics, demonstrating the key ideas with simple examples. The first part of the chapter covers Delphi's integrated debugger and various features ET provides for runtime debugging. Then, I'll describe some other debugging techniques and discuss how you can monitor the flow of messages in your application. The final section describes how you can examine the status of the memory used by a program.

Functions for start and debugging script are shown in the picture below:



This button execute the script



This button pause the script



This button stop script execution



Trace Into. With this button you can debug a specific Function.



Trace Out. If the cursor is positioned on a function call, you can go on without enter in the function's code with this button



Insert breakpoint

As the name implies, a breakpoint, when reached, is supposed to stop the program execution. In ET, breakpoints can do more than just stop. Each breakpoint can have any of several actions associated with it. These actions can be the traditional break action, the display of a fixed string or a calculated expression in the message log, or the activation or deactivation of other groups of breakpoints.

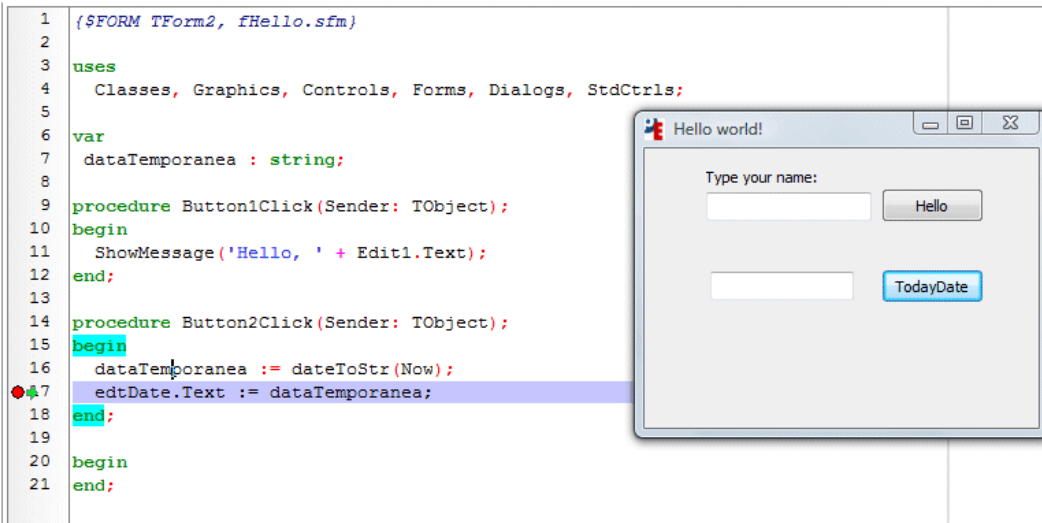
You can insert a breakpoint in the code before start execution with a simple click. Position the cursor on the left of number instruction then click left button mouse. A red ball will appears.

```

4   Classes, Graphics, Controls, Forms, Dialogs, St
5
6   var
7     dataTemporanea : string;
8
9   procedure Button1Click(Sender: TObject);
10  begin
11    ShowMessage('Hello, ' + Edit1.Text);
12  end;
13
14  procedure Button2Click(Sender: TObject);
15  begin
16    dataTemporanea := dateToStr(Now);
17    edtDate.Text := dataTemporanea;
18  end;
19
20  begin
21  end;

```

if you run the script, when the code has to execute line 13 the RAD stops execution and show the code:



3 Buttons allow you to watch expression during debug session:



Enable and disable Watch windows

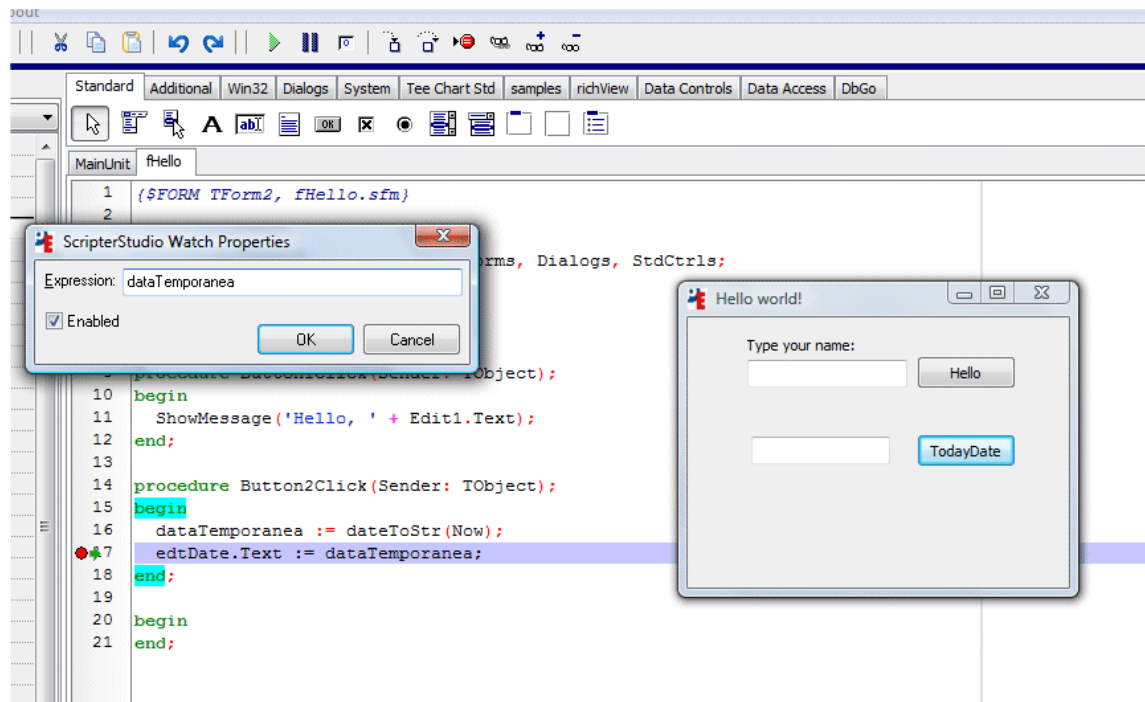


Add Watch Expression

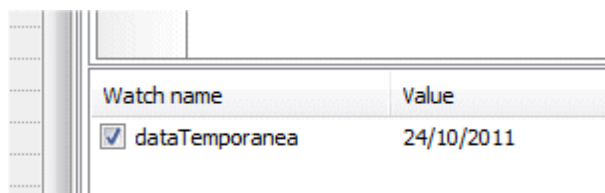


Remove Selected Watch Expression

Clicking now the add watch expression button a windows ask you to insert the variable name you want inspect



you can see watch expressions list in the windows under the code. If you don't see that windows you must click **Enable and disable Watch windows** button



4.7 Understanding the Script unit source

Navigation: Using ET RAD >

Understanding the Delphi unit source

Understanding the unit source

Forms are visible building blocks of all (well, at least 99%) Delphi projects. Each form in a Delphi project has an associated unit. The unit contains the source code for any event handlers attached to the events of the form or the components it contains.

The best way to describe the unit code is to take a look at the source. For the moment refer to the example in the last chapter, especially the unit source. After we have placed a Label, an Edit box and a Button, and added an OnClick event handling procedure for the button, the source code looked like:

```
01: unit Unit1;
02: interface
03: uses
03: Windows, Messages, SysUtils, Variants, Classes,
03: Graphics, Controls, Forms, Dialogs, StdCtrls;
```

```
04: type
05:   TForm1 = class(TForm)
06:     Edit1: TEdit;
07:     Button1: TButton;
08:     Label1: TLabel;
09:     procedure Button1Click(Sender: TObject);
10:   private
11:     { Private declarations }
12:   public
13:     { Public declarations }
14:   end;

15: var
16:   Form1: TForm1;

17: implementation
18:

19: procedure TForm1.Button1Click(Sender: TObject);
20: var s: string;
21: begin
22:   s := 'Hello ' + Edit1.Text + ' Delphi welcomes you!';
23:   ShowMessage(s);
24: end;

25: end.
```

The UNIT keyword

A unit file begins with a unit heading, which is followed by the interface, implementation, initialization, and finalization sections. The [initialization and finalization](#) sections are optional.

The unit heading starts with a word unit (line 01), followed by a unit (file) name. The unit name (Unit1 in the above source) must match the unit file name on a disk. In a single project all unit names must be unique. You should change the unit's name only by using the File-Save As command from the Delphi IDE main menu. Of course, it is completely up to you to decide how will you name your units. In most cases you'll want your units to have the name similar to the name of the form to which they are linked, like 'MainUnit' for Main form (form with a Name property set to 'Main'). Be sure to give name to units in the early stage of a form design development.

The INTERFACE section

The interface section of a unit starts with the word interface (line 02) and continues until the word implementation (line 17). This section is used to declare any [public](#) sections of code that appear in a unit. The entire contents of the interface section, including type, variable and procedure declarations, are visible to any other unit which uses this unit. When any other part of the program looks at a unit, all it sees is the interface section. Everything else is hidden, internal to the unit, part of the implementation. You could say that the interface section contains a list of items in the unit that other units can use.

In most cases the interface section will define several "subsections", you can see that the code for unit1.pas has a uses clause, a type section, and a variable declaration section.

The INTERFACE USES section

If the interface section includes a uses clause, it must appear immediately after the

word interface. A uses clause (line 03) lists units used by the unit. In most cases, all necessary units are placed in the interface uses clause when Delphi compiler generates and maintains a units source. The Windows, Messages, SysUtils, etc are all standard Delphi units, required by a program.

As you drop components on a form, the necessary units will be added automatically to the uses clause. For example, if you add a TOpenDialog component on your form (Dialogs page on the component palette), the Dialogs unit will appear in the uses clause because it contains the logic for the TOpenDialog component (and other Dialog components).

In some situations, you'll need to manually add units to interface uses clause. Suppose you are to use the TRegistry object, designed to access the Windows Registry. You cannot drop the TRegistry component on a form, since it does not appear on the component palette - you must manually add the word Registry to the uses list.

The INTERFACE TYPE section

Another part of the interface section is the type section. The form type declaration (or form class declaration) section introduces the form as a class. The code from line 04 to 14 declares the existence and structure of a class called TForm1.

A few words on classes and objects

I'm aware that this is not the place to explain OOP in Delphi, but I sense that something must be stated. The basics of object oriented programming in Delphi will be discussed in the next chapter of this course, however some words must be explained now.

A class, or class type, defines a structure consisting of fields, methods, and properties. Instances of a class type are called objects.

For example, in real world, a class PROGRAMMER can have properties like: Years_Of_Experience and Projects_Developed. It can expose methods like: Write_Program and Talk_To_Users. A class is something that does not truly exist. An object: DELPHI PROGRAMMER is a specific instance of a class.

The TForm1 is a class inherited from TForm (line 05).

Each component dropped on a form becomes a field (or variable) of the TForm1 class (lines 06 through 08). For example, Edit1 is a variable of a TEdit type, which you see on the screen when you run the program. When you need to read a value from this particular edit box you use the Edit1 variable, like in 's := Edit1.Text'.

Each event handling procedure for a form events or events for components dropped on a form (form fields) will have its declaration (line 09) in the interface type part.

The INTERFACE VAR section

This part (line 15,16) of the interface section is used to declare (create) a Form1 object as an instance of the TForm1 class. If you have created your own data type (with fields, properties and methods) as a part of this unit, you could create a variable of that type in this part of the interface section.

The IMPLEMENTATION section

The implementation section is defined as everything between the implementation word and either the initialization statement or the end of the file (as denoted by the end. keyword).

The implementation is where you write code that performs actions. This section is private to the unit, and can contain both declarations and code. The implementation section of a unit can contain its own uses clause as well.

A few words on using another unit (form)

As you will see in the following chapters of this course, a (form) unit can use another unit. Simply put, this means that one form can call another form. Suppose you have a main form (form name: MainForm, unit name:

MainFormUnit) in a project with an 'About...' button on it. What you want to do is to show an about box form (form name: AboutForm, unit name: AboutFormUnit) when you click on this button. To be able to do this the MainFormUnit must use the AboutFormUnit, the AboutFormUnit should be placed in the implementation uses clause.

To actually call a method (procedure or function) from MainFormUnit that is declared in the AboutFormUnit, you use the following syntax:

```
AboutFormUnit.SomeProcedureName(parameters)
```

Note that the call to a procedure SomeProcedureName consists of a unit name (AboutFormUnit) followed by a period (.) and a procedure name. This fact is very important. If in some stage of the development of your project you decide to save AboutFormUnit under a different name - you will need to change the call to any procedure inside that unit, since the name of the unit will no longer be AboutFormUnit. This is the reason why you should give meaningful name to units in the early stage of form (unit) development.

Anything that appears in a unit's implementation section that is not referenced in the interface is private to that unit. This means that a procedure or function declared and defined (implemented) in the implementation section cannot be called from another unit unless its header is listed in that unit's interface.

The **INITIALIZATION** and **FINALIZATION** sections

These two sections are optional; they are not automatically generated when we create a unit. If we want to initialize any data the unit uses, we can add an initialization code to the initialization section of the unit. When an application uses a unit, the code within the unit's initialization part is called before the any other application code runs.

If your unit needs to perform any cleanup when the application terminates, such as freeing any resources allocated in the initialization part; you can add a finalization section to your unit. The finalization section comes after the initialization section, but before the final end.

4.8 Standard tab GUI components

Navigation: Using ET RAD >

Standard tab GUI components

GUI components GUI stands for Graphical User Interface. It refers to the windows, buttons, dialogs, menus and everything visual in a modern application. A GUI component is one of these graphical building blocks. Delphi lets you build powerful applications using a rich variety of these components.

These components are grouped under a long set of tabs in the top part of the Delphi screen, starting with **Standard** at the left. We'll look at this Standard tab here. It looks something like this (Delphi allows you to tinker with nearly everything in its interface, so it may look different on your system):



Each of the components is described below with a picture of a typical GUI object they can create:

Note that the displayed components were taken from an XP computer. In order to get the new XP look (the XP 'themed' GUI look), you must add the **XP Manifest** component to your form. It is found under the **Win32** component tab:



Menus After you add a TMenu component to your form, you can design the menu by double clicking it (or using the right button popup menu for it). You are then shown a panel with an empty menu. As you type, you are creating the top left menu item. Press enter and you are positioned at the first sub item of this menu item. Click the new empty box to the right of the first menu item to create a new menu item. In this way, you can build the menu structure. To make each menu item do something, just double click it. Delphi will then insert code into your program to handle the menu item, and position your cursor in the form unit ready for you to write your code. Explore the popup menu for the menu editor to discover more options, such as sub-menus. A menu can also be dynamically updated by your code.



Popup menus A popup menu appears in many applications when you right click on something. For example, when you right click the Windows desktop. You create a popup menu by adding the popup menu component to your form and double clicking it. You then simply type in your menu item list. You attach the popup menu to an existing form object (or the form itself) by selecting your new popup menu in the **PopupMenu** property of the object. To activate the popup menu items, double click each in turn. Delphi will add the appropriate code to your form unit. You can then type in the code that each menu item should perform. A popup menu can also be dynamically updated by your code.



Labels Labels are the simplest component. They are used to literally label things on a form, but the text, colours and so on can be changed by your code. For example, you can change the label colour when the mouse hovers over it, and can run code when the user clicks it. This makes the label like a web page link. Normally, they are just kept as plain, unchanging text.



Edit boxes An edit box allows the user to type in a single line of text. For example, the name of the user. You set up the initial value with the **Text** property either at design time or when your code runs.



Memo boxes A memo box displays a single string as a multi line wrapped text display. You cannot apply any formatting. The displayed lines are set using the **Lines** property. This may be set at design time as well as at run time.



Buttons A button is the simplest active item. When clicked by a user, it performs some action. You can change the button label by setting the **Caption** property. Double clicking the button when designing adds code to your form to run

when the button is clicked at run time.



Check boxes Check boxes are used to give a user a **yes/no** choice. For example, whether to wrap text or not. The label is set using the **Caption** property. You can preset the check box to ticked by setting the **Checked** property to **true**.



Radio buttons Radio buttons are used to give a user **multiple** choices. For example, whether to left, centre or right align text. The label is set using the **Caption** property. You can preset a radio button to selected by setting the **Checked** property to **true**. You would normally use radio buttons in groups of two or more. The **TRadioGroup** component allows you to do this in a neat and dynamic way.



List boxes List boxes provide selectable items. For example, a collection of fish names. If you set the **MultiSelect** property to **true**, you allow the user to select more than one. The items in the list are added using the **Items.Add** method, passing the string of each item as a parameter. You can act upon an item being selected by setting the **OnClick** event (by double clicking it) to a procedure in your form unit. The following example displays the selected list item in a dialog box:

```
procedure TForm1.ListBox1Click(Sender: TObject);
var
  listBox : TListBox;
  index : Integer;
begin
  // Cast the passed object to its correct type
  listBox := TListBox(Sender);

  // Get the index of the selected list item
  index := listBox.ItemIndex;

  // Display the selected list item value
  ShowMessage(listBox.Items[index]);
end;
```



Combo boxes A combo box is like a list box, and is set up in the same way (see above). It just takes up less space on your form by collapsing to a single line when deselected, showing the chosen list item. It is not recommended to use one for multi line selection.



Scroll bars Many components have built in scroll bars. For those that don't, you can use this to do your own scrolling. You link the scrollbar to your component by setting the **OnScroll** event. This gives you the details of the last scroll activity made by the user.

Group boxes A group box is like a panel. It differs in that it gives a name to the collection of components that you add to it. This title is set with the **Caption** property. Use a group box to help the user see what controls affect one particular

aspect of the application.



Radio group panels Radio buttons are used to give a user a **multiple** choices. For example, whether to left, centre or right align text. Unlike individual radio buttons, a group is only set up by your code. You define the buttons by calling the **Items.Add** method of the TRadioGroup object, passing the caption string of each radio button as a parameter. You can reference each button by using the **Buttons** indexed property. You might, for example, choose the third button to be checked. For example :

```
// Set the third button to be pre-selected (index starts at 0)
RadioGroup1.Buttons[2].Checked := true;
```



Empty panels When building your form, you might want to add many components. These may fall into logical groups. If so, you can add each group to a panel, and use the panel to position the whole group on the form. The panel name can be blanked out by setting the **Caption** property. You can even hide the panel by setting the **BevelOuter** and **BevelInner** properties to **bvNone**.

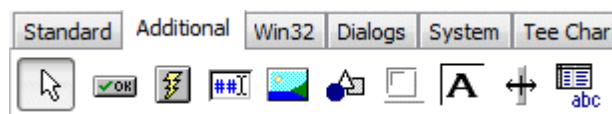
4.9 Additional tab GUI components

Navigation: Using ET RAD >

Additional tab GUI components

GUI components GUI stands for Graphical User Interface. It refers to the windows, buttons, dialogs, menus and everything visual in a modern application. A GUI component is one of these graphical building blocks. Delphi lets you build powerful applications using a rich variety of these components.

The second component group is **Additional**



Each of the components is described below with a picture of a typical GUI object they can create:



Two types of Buttons A button is the simplest active item. When clicked by a user, it performs some action. You can change the button label by setting the **Caption** property. Double clicking the button when designing adds code to your form to run when the button is clicked at run time.



Mask Edit boxes An edit box allows the user to type in a single line of text.

For example, the name of the user. You set up the initial value with the **Text** property either at design time or when your code runs. This component has a property Editmask which permits a controlled data entry



Image ET's image component, TImage, displays a graphical image, like a bitmap, icon, or metafile. Properties and methods of can be used for such things as loading an image from file, clearing the image in the TImage, and assigning an image for another control. Learn how to use the image control in Delphi programs



Shape With that component you can design geometrical figures in the form



Static Labels Labels are the simplest component. They are used to literally label things on a form, but the text, colours and so on can be changed by your code. For example, you can change the label colour when the mouse hovers over it, and can run code when the user clicks it. This makes the label like a web page link. Normally, they are just kept as plain, unchanging text.



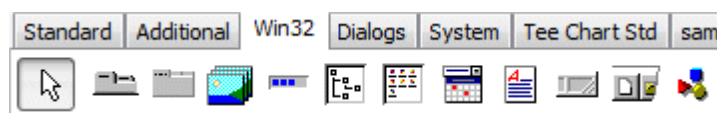
String Grid

4.10 Win32 tab GUI components

Navigation: Using ET RAD >

Win32 tab GUI components

GUI components GUI stands for Graphical User Interface. It refers to the windows, buttons, dialogs, menus and everything visual in a modern application. A GUI component is one of these graphical building blocks. Delphi lets you build powerful applications using a rich variety of these components. The third component group is **Win32**



Each of the components is described below with a picture of a typical GUI object they can create:



Tab and Page control



Progress Bar When your application performs a time-consuming operation, you can use a progress bar, the TProgressBar Delphi control, to show how much of the task is completed.



Tree view The TTreeView Delphi component represents a window that displays

a hierarchical list of items, such as the headings in a document, the entries in an index, or the files and directories on a disk.



List view The TListView Delphi control displays a list of items in a fashion similar to how Windows Explorer displays files and folders. Items can be displayed in columns with column headers and sub-items, or vertically or horizontally, with small or large icons. Learn how to use the list view control in Delphi programs



DateTime picker component for setting date and time.



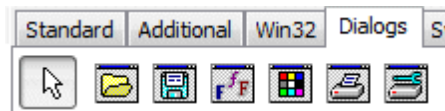
Rich edit TRichEdit is a memo control that supports rich text formatting. Learn how to use various multi-line text controls in Delphi's VCL.

4.11 Dialog tab GUI components

Navigation: Using ET RAD >

Dialog tab GUI components

GUI components GUI stands for Graphical User Interface. It refers to the windows, buttons, dialogs, menus and everything visual in a modern application. A GUI component is one of these graphical building blocks. Delphi lets you build powerful applications using a rich variety of these components. The fourth component group is **Dialogs**



Set of component which display a dialog form which permits you to:



open a file Open Dialog Component



save a file Save Dialog Component



select a font Font Dialog Component



select a color Color Dialog Component



set and start a print and printer manager Print Dialog Component

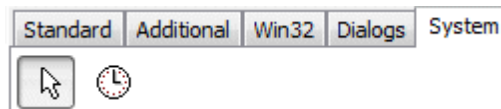
4.12 System tab GUI components

Navigation: Using ET RAD >

Dialog tab GUI components

GUI components GUI stands for Graphical User Interface. It refers to the windows, buttons, dialogs, menus and everything visual in a modern application. A GUI component is one of these graphical building blocks. Delphi lets you build powerful applications using a rich variety of these components.

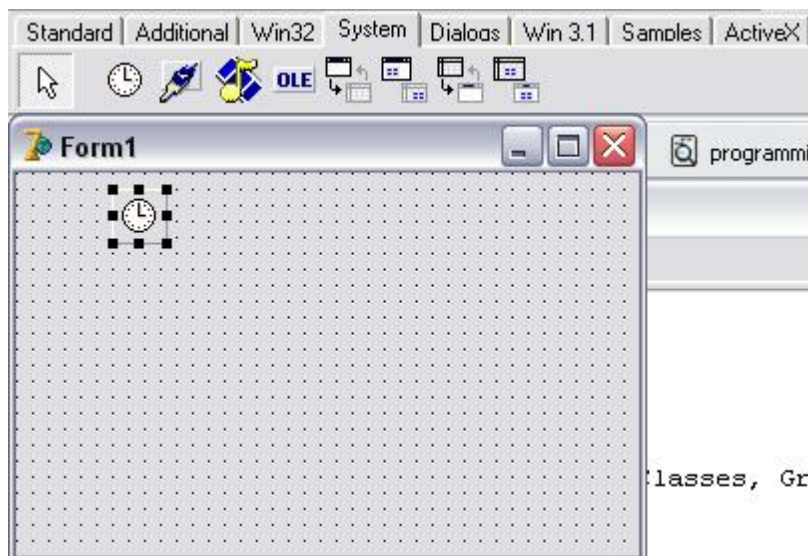
The fifth component group is System and contains only one Componente



TTimer

How to use the TTimer in ET?

Add this on your form. Like this:



Now look at the **Object Inspector**, you'll see this:



The "interval" means how many seconds eg. you should wait or something like that. 1000 means 1 second. So if you type 5000 it means 5 seconds, get it? good

Now try to type 10000 in the interval box. And double click on the TTimer you just added. Now the code will show.

In the code try to add this:

Code:

```
Showmessage('You just created your first application with TTimer!');
```

After 10 seconds a pop-up will appear. This means you created your first application with TTimer!

4.13 Open Dialog Component

Navigation: Using ET RAD >

Open Dialog Component

Description

The **Open Dialog** is a visual component imported from . It is used to allow a user to select one or more files to open.

It can be defined by dragging the open dialog icon from the Dialogs tab in ET, or by defining a TOpenDialog variable.

The **TOpenDialog** can be configured to suit your needs. When using it, you would proceed along the following steps:

Creating the dialog object

You define a TOpenDialog variable, and then assign a new TOpenDialog object to it:

```
var  
openDialog : TOpenDialog;  
begin  
openDialog := TOpenDialog.Create(self);
```

Note that the dialog must have an anchor - here we provide the current object - self - as the anchor.

Setting options

Before displaying the dialog, you are likely to configure it to your needs by setting the dialog properties. Here are the main properties:

Title property

Used to set the caption for the dialog.

FileName property

Gives a default file name to open. (Otherwise, the file name field is blank). When returning from the dialog, if the user has hit OK, this property will contain the (first) selected file name, including its full path (see the first example).

Filter property

This allows only certain file types to be displayed and selectable. The filter text is displayed in a drop down below the file name field. The following example selects for text files only:

```
openDialog.Filter := 'Text files only|*.txt';
```

The drop down dialog shows the description before the | separator. After the separator, you define a mask that selects the files you want.

```
openDialog.Filter := 'Text and Word files only|*.txt;*.doc';
```

Above we have allowed two different file types, separated by a ;.

```
openDialog.Filter := 'Text files|*.txt|Word files|*.doc';
```

Above we have allowed text and Word files as two options in the drop down list.

FilterIndex property

Defines which (starting at 1) of the drop down filter choices will be displayed first.

InitialDir property

Sets the starting directory in the dialog.

Options property

This is a set of TOpenOptions flags. These are quite extensive. The key values are:

ofReadOnly Opens the file for read only

ofFileMustExist Only existing file may be opened

ofAllowMultiSelect User can select 2 or more files

Displaying the dialog

We now call a method of TOpenDialog:

```
if openDialog.Execute  
then ...
```

Execute returns **true** if the user selected a file and hit **OK**. You can then use the selected file:

Finishing with the dialog

The selected file or files are obtained using the following properties:

FileName property

This holds the full path plus file name of the selected file

Files property

This holds the full path plus file name of the a multiple file selection. The file names are held in the returned **TStrings** value (see the **TStringList** for more on string lists).

Finally, we must free the dialog object:

```
openDialog.free;
```

Example code : Illustrating single file selection var

```
openDialog : TOpenDialog; // Open dialog variable
begin
// Create the open dialog object - assign to our open dialog variable
openDialog := TOpenDialog.Create(self);

// Set up the starting directory to be the current one
openDialog.InitialDir := GetCurrentDir;

// Only allow existing files to be selected
openDialog.Options := [ofFileMustExist];

// Allow only .dpr and .pas files to be selected
openDialog.Filter :=
'Delphi project files|*.dpr|Delphi pascal files|*.pas';

// Select pascal files as the starting filter type
openDialog.FilterIndex := 2;

// Display the open file dialog
if openDialog.Execute
then ShowMessage('File : '+openDialog.FileName)
else ShowMessage('Open file was cancelled');

// Free up the dialog
openDialog.Free;
end;
```

4.14 Save Dialog Component

Navigation: Using ET RAD >

Save Dialog Component

The **Save Dialog** is a visual component. It is used to allow a user to select the name of a file to save to.

It can be defined by dragging the save dialog icon from the Dialogs tab in Delphi, or by defining a TSaveDialog variable.

The **TSaveDialog** can be configured to suit your needs. When using it, you would proceed along the following steps:

Creating the dialog object

You define a TSaveDialog variable, and then assign a new TSaveDialog object to it:

```
var  
saveDialog : TSaveDialog;  
begin  
saveDialog := TSaveDialog.Create(self);
```

Note that the dialog must have an anchor - here we provide the current object - self - as the anchor.

Setting options

Before displaying the dialog, you are likely to configure it to your needs by setting the dialog properties. Here are the main properties:

Title property

Gives the caption to the dialog.

FileName property

Gives a default file name to save. (Otherwise, the file name field is blank).

DefaultExt property

Defines the extension that will be added to the user file name, if manually typed (rather than selected from the file list). If there are two or more save filter extension types, then this value is ignored. However, it must be provided in order for the drop down list extension values to be used. Strange!

Filter property

This allows only certain file types to be displayed and selectable. The filter text is displayed in a drop down below the file name field. The following example selects for text files only:

```
saveDialog.Filter := 'Text files only|*.txt';
```

The drop down dialog shows the description before the | separator. After the separator, you define a mask that selects the files you want.

```
saveDialog.Filter := 'Text files|*.txt|Word files|*.doc';
```

Above we have allowed text and Word files as two options in the drop down list.

FilterIndex property

Defines which (starting at 1) of the drop down filter choices will be displayed first.

InitialDir property

Sets the starting directory in the dialog.

Displaying the dialog

We now call a method of TSaveDialog:

```
if saveDialog.Execute  
then ...
```

Execute returns **true** if the user selected a file and hit **OK**. You can then save to the selected file:

Finishing with the dialog

The selected file obtained using the following property:

FileName property

This holds the full path plus file name of the selected file. Finally, we must free the dialog object:

```
saveDialog.free;
```

4.15 Font Dialog Component

Navigation: Using ET RAD >

Font Dialog Component

To use the **Font dialog box**, the user should first have text that needs to, and can, be formatted. Users usually call the Font dialog box using a menu item or a popup menu from right clicking. Once the dialog box displays, a user can select a font by its name, its style, its size, one or both effects (Underline or Strikeout), and a color. After making the necessary changes, the user can click OK to apply the changes or click Cancel to ignore the selected attributes

At design time, the Font dialog box hardly needs any change of properties to [work](#). The only time you would set its properties is if you judge that its default properties are not conform to your particular scenario. For example, if you are providing font formatting for a control and you want users to control the font characteristics of individual letters or paragraph, there should be nothing to change at design time. Otherwise, the default properties can be changed using the Object Inspector.

Once, and however, you have a **TFontDialog** instance, you can display the Font dialog box by calling the **Execute()** method. The font dialog box is equipped with two primary buttons: OK and Cancel. After using it, if the user clicks OK, this implies that if there were changes of font, size, color, etc, the user wants them committed to the document. If the user clicks Cancel, this means that you should ignore any actions that were performed on the dialog box. The **Execute()** method is Boolean. It returns true if the user clicks OK. Otherwise, if the user clicks Cancel, it would return false. Therefore, after the user has used it, you should find out if she clicked OK or Cancel before applying her changes. This inquiry is usually performed with an if conditional statement as follows:

```
procedure TForm1.btnFontClick(Sender: TObject);  
var dlgFont :  
    FontDialog;  
begin
```

```
dlgFont := TFontDialog.Create(Form1);
if dlgFont.Execute then // What to do if the user clicked OK
end;
```

The Font From the Dialog Box

As its name indicates, the Font dialog box is used to configure or get a font. The font of the dialog box is an object of type **TFont**. To support it, the **TFontDialog** class is equipped with a property named **Font** and that is of type **TFont**:

```
property Font: TFont read FFont write SetFont;
```

After the user has used a Font dialog box and clicked OK, you can find out what font was selected and assign it to the **TFont** object that needs it or to assign it to the **Font** property of the object you are using. Here is an example:

```
procedure TForm1.btnFontClick(Sender: TObject);
var
  dlgFont : TFontDialog;
begin
  dlgFont := TFontDialog.Create(Form1);

  if dlgFont.Execute then
    Font.Name := dlgFont.Font.Name;
end;
```

The styles can be managed using the Font dialog box as one object. After the user has clicked OK on the dialog box, you can simply assign whatever style was set to the **TFont.Style** property of the object that needs the change. Here is an example:

```
procedure TForm1.btnFontClick(Sender: TObject);
var
  dlgFont : TFontDialog;
begin
  dlgFont := TFontDialog.Create(Form1);

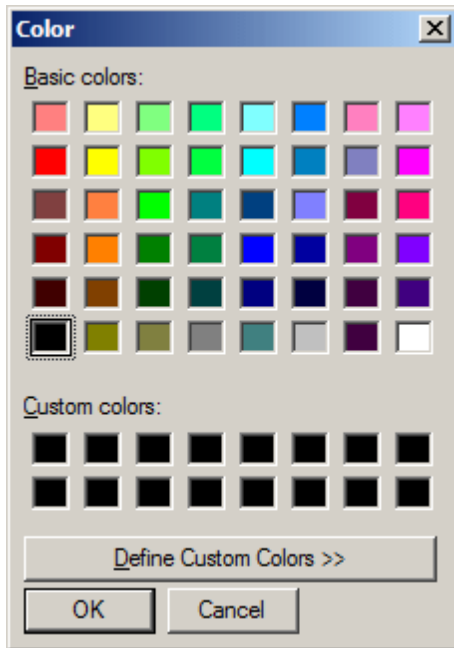
  if dlgFont.Execute then
  begin
    Font.Name := dlgFont.Font.Name;
    Font.Size := dlgFont.Font.Size;
    Font.Color := dlgFont.Font.Color;
    Font.Style := dlgFont.Font.Style;
  end
end;
```

4.16 Color Dialog Component

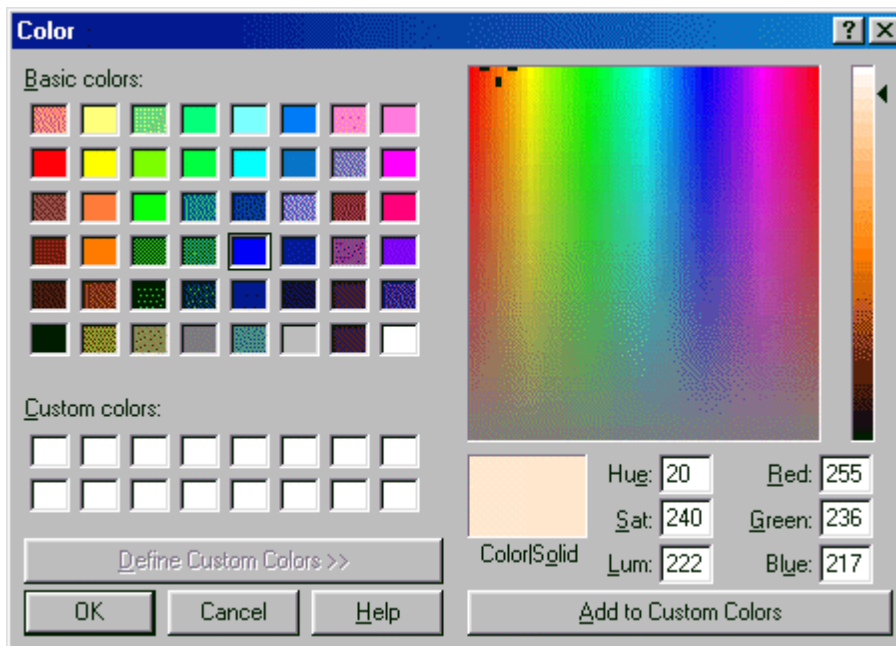
Navigation: Using ET RAD >

Color Dialog Component

To provide the selection of colors on Microsoft Windows applications, the operating [system](#) provides a common dialog box appropriate for such tasks. The **Color dialog box** is used by various reasons to let the user set or change a color of an object such as the [background color](#) of a control or the color used to [paint](#) an object. When it displays, by default, the dialog box appears as follows:



This displays a constant list of colors to the user. If none of the available colors is appropriate for the task at hand, the user can click the Define Custom Colors button to expand the dialog box:



The expanded Color dialog box allows the user to either select one of the preset colors or to custom create a color by specifying its red, green, and blue values.

The user can change the color in four different areas. The top left section displays a list of 48 predefined colors. If the desired color is not in that section, the user can click and drag the mouse in the multi-colored palette. The user can also drag the right bar that displays a range based on the color of the palette; the user can scroll up and down by dragging the arrow. For more precision, the user can type the Red, Green and Blue values. Each uses a integral value that ranges from 1 to 255.

Creating a Color Dialog Box

In the VCL, the color dialog box is available through a class named **TColorDialog**.

The **TColorDialog** class is derived from the **TCommonDialog** class. The **TCommonDialog** class is derived from **TComponent**.

To visually add a Color dialog box to your application, from the Dialogs section of the

Tool Palette, click the TColorDialog button  and click anywhere on the form. To programmatically create a color dialog box, declare a variable of type **TColorDialog**. Here is an example:

```
procedure TForm1.btnColorClick(Sender: TObject);
var Dlg : TColorDialog;
begin
  Dlg := TColorDialog.Create(Form1);
end;
```

Characteristics of the Color Dialog Box

The Color

The most important and most obvious property of the Color dialog box is the selected color once the user has made a choice. To provide this information, the **TColorDialog** class is equipped with the **Color** property:

property Color: TColor read FColor write FColor;

When the user opens the dialog, you can set the default color on the Object Inspector using the Color property. You can also set this color programmatically as follows:

```
procedure TForm1.btnColorClick(Sender: TObject);
var Dlg : TColorDialog;
begin
  Dlg := TColorDialog.Create(Form1);
  Dlg.Color := clRed;
end;
```

When the user has finished using the Color dialog box and clicks OK, you can find out what color was selected by using the **TColorDialog.Color** property.

The Size of the Dialog Box

You can control the regular or full size of the dialog using the Options property:

property Options: TColorDialogOptions read FOptions write FOptions;

At design time, to manipulate the options, on the Object Inspector, click the + button on the Options field to expand it. Since the options are controlled by the **TColorDialogOption** is a set, you can specify as many options as you want:

```
procedure TForm1.btnColorClick(Sender: TObject);
var Dlg : TColorDialog;
begin
  Dlg := TColorDialog.Create(Form1);
  Dlg.Color := clRed;

  Dlg.Options := [cdFullOpen, cdAnyColor];
end;
```

4.17 Print Dialog Component

Navigation: Using ET RAD >

Print Dialog Component

The **Print Dialog Component** is used to create a printer selection and print control dialog.

Before printing from your application, it is wise to display a print dialog. This allows the user to select the desired printer and attributes, along with control over how the document is printed. Such as multiple copies and pages to be printed.

You first use the class by creating an object from it, and then setting the required dialog attributes from the following list:

- Collate** Whether to preset the Collate option
- Copies** How many copies to print
- FromPage** Selected page in page range
- ToPage** Selected page in page range
- MinPage** Earliest selectable page
- MaxPage** Latest selectable page
- PrintRange** Starting page selection type
- Options** Various multi-selectable options
- PrintToFile** If false, we print to paper

The **PrintRange** values are:

- prAllPages** All pages to print
- prSelection** Page selection to print
- prPageNums** Page number range to print

You choose one before the dialog starts - and check to see if the user has changed it when the dialog ends.

The **Options** values may be one of the following:

- poPrintToFile** Print to file
- poPageNums** Print by page range
- poSelection** Print by page selection

poWarningWarning if bad printer

poHelpDisply help

poDisablePrintToFilePrint to file disallowed

Some of these options may also be set by the dialog user. Always check their values afterwards.

After setting these options, use the **Execute** method to display the dialog, checking the Boolean outcome after it runs to know whether to proceed with the printing.

Example code :

```

const
TOTAL_PAGES = 4; // How many pages to print
var
printDialog : TPrintDialog;
page, startPage, endPage : Integer;

begin
// Create a printer selection dialog
printDialog := TPrintDialog.Create(Form1);

// Set up print dialog options
printDialog.MinPage := 1; // First allowed page number
printDialog.MaxPage := TOTAL_PAGES; // Highest allowed page number
printDialog.ToPage := TOTAL_PAGES; // 1 to ToPage page range allowed
printDialog.Options := [poPageNums]; // Allow page range selection

// If the user has selected a printer (or default), then print!
if printDialog.Execute then
begin
// Use the Printer function to get access to the global TPrinter object.
// Set to landscape orientation
Printer.Orientation := poLandscape;

// Set the printjob title - as it it appears in the print job manager
Printer.Title := 'Test print for Delphi';

// Set the number of copies to print each page
// This is crude - it doies not take Collation into account
Printer.Copies := printDialog.Copies;

// Start printing
Printer.BeginDoc;

// Finish printing
Printer.EndDoc;
end;
end;

```

4.18 Learn about: properties, events and ET RAD

Navigation: Using ET RAD >

Learn about: properties, events and ET RAD

Changing Component Properties

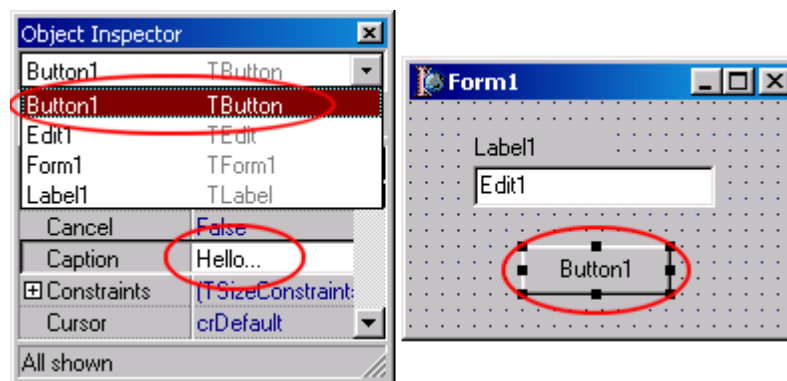
After you place components on a form, you can set their properties with the Object Inspector. The properties are different for each type of component, some properties apply to most components. Altering a component property, changes the way a component behaves and appears in an application.

All the components have a property called "Name". The Name property is very important; it specifies the name of the component as referenced in code. When you first place a component on a form, Delphi will provide a default name for the component: Label1, Edit1, Button1. I suggest you to give your components a meaningful name before writing the code that refers to them. You can do this by changing the value of the Name property in the Object Inspector.

Note: with the last statement in mind, I'll do the opposite. In most cases, I'll leave all the default component names through this Course - just as they appear when you place them on a form.

To actually change a component property you first need to activate it - click it to select it - small square handles appear at each corner and in the middle of each side. Another way to select a component is to click its name in the drop down list that appears at the top of the Object Inspector. This list lists all the components on the active form along with their types in the following format: "Name Type".

When a component is selected, its properties (and events) are displayed in the Object Inspector. To change the component property click on a property name in the Object Inspector; then either type a new value or select from the drop-down list. For example, change the Caption property for Button1 (I'll refer components by their names) to 'Hello...' (of course without the single quotation marks)



Components have different kinds of properties; some can store a boolean value (True or False), like Enabled. To change a boolean property double click the property value to toggle between the states. Some properties can hold a number (e.g. Width or Left), a string (e.g. Caption or Text) or even a set of "simple valued" properties. When a property has an associated editor, to set complex values, an ellipsis button appears near the property name. For example if you click the ellipsis of the Font property a Font property dialog box will appear.

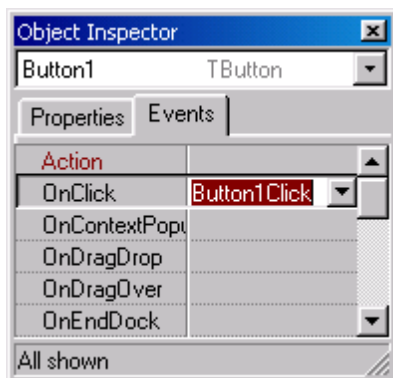
Now, change the Caption (the static text the label displays on the form) of Label1 to 'Your name please:'. Change the Text property (text displayed in the edit box - this text will be changeable at run time) of Edit1 to 'Prova'

Writing Code - Events and Event Handlers

To really enable components to do something meaningful you have to write some

action-specific code for each component you want to react on user input. Remember: components are building block of any Delphi form, the code behind each component ensures a component will react on an action.

Each Delphi component, beside its properties, has a set of events. Windows as even-
lled environment requires the programmer to decide how a program will (if it will) react
on user actions. You need to understand that Windows is a message-based
operating system. System messages are handled by a message handler that
translates the message to Delphi event handlers. For instance, when a user clicks a
button on a form, Windows sends a message to the application and the application
reacts to this new event. If the OnClick event for a button is specified it gets
executed.



The code to respond to events is contained in Delphi event procedures (event handlers). All components have a set of events that they can react on. For example, all clickable components have an OnClick event that gets fired if a user clicks a component with a mouse. All such components have an event for getting and loosing the focus, too. However if you do not specify the code for OnEnter and OnExit (OnEnter - got focus; OnExit - lost focus) the event will be ignored by your application.

To see a list of events a component can react on, select a component and in the Object Inspector activate the Events tab. To really create an event handling procedure, decide on what event you want your component to react, and double click the event name.

For example, select the Button1 component, and double click the OnClick event name. Delphi will bring the Code Editor to the top of the screen and the skeleton code for the OnClick event will be created.

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  //this is where your code goes  
end
```

Note: For the moment there is no need to understand what all the words in the above code stand for. Just follow along, we'll explain all that in the following chapters.

As you will understand more clearly through this course, a procedure must have a unique name within the form. The above procedure, Delphi component event-driven procedure, is named for you. The name consists of: the name of the form (prefixed with T) "TForm", a full stop ".", the component name "Button1", and the event name "Click". For any component there is a set of events that you could create event

handlers for. Just creating an event handler does not guarantee your application will do something on the event - you must write some event handling code in the body of the procedure.

A few words on Delphi (Object) Pascal

The code you write inside event procedures is Pascal code. Object Pascal or Delphi Pascal (as I will mostly call it), a set of object-oriented extensions to standard Pascal, is the language of Delphi. Delphi Pascal enables you to take advantage of object-oriented programming to its fullest. It can be said that Delphi Pascal is to Pascal what C++ is to C. As Delphi was being developed, new language behavior and keywords were added to deal with the component model. In general, Delphi Pascal is a high-level, compiled, strongly typed language that supports structured and object-oriented design.

We'll now write some code for the OnClick event handler of Button1. Alter the above procedure body to:

```
procedure TForm1.Button1Click(Sender: TObject);  
var s: string;  
begin  
  s := 'Hello ' + Edit1.Text + ' ET welcomes you!';  
  ShowMessage(s);  
end;
```

4.19 Code Samples

4.19.1 3 MURI OP

Navigation: Using ET RAD > Code Samples >

3 MURI OP

a good way to create a new project starting from Script_op project is as follow:

1. copy all files in \Script\TreMuriDataExport into a new folder of you choice
2. open the project Script_op.ssproj from this new folder
3. from menu File > Save Project as > save the project with a new name of you choice
4. Script_OP code must not be modified

```
// *****  
// Modulo standard contenente dichiarazioni e chiamata a form Principale  
// NON deve essere modificato
```

```
// *****
```

```
uses
```

```
Classes, Graphics, Controls, Forms, Dialogs, DemoOPMain, OPFunctions;
```

```
var
```

```
MainForm: TfrmMain3mSample;
```

```
begin
```

```
MainForm := TfrmMain3mSample.Create(Application);
```

```
MainForm.Show;
```

```
end;
```

5. OPFunctions is the include containing declaration's function to interface 3M OP and you have not to modify it. For a detailed explanations about each functions see 3Muri-OP tutorial

6. works on DemoOPMain. you can modify code and form as you wish.. the code already present will help you in developing your owns functions and procedures

4.19.1.1 3 MURI OP Interface

Navigation: Using ET RAD > Code Samples > 3 MURI OP >

3 MURI OP Interface

```
// *****
```

```
// Dichiarazione interfacce
```

```
// *****
```

```
function selectDirectory(dirStart:string) : String external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_selectDirectory';
```

```
function listFileFromDirectory(normativa: string; dirStart:string; mask:string) : TstringList external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3Muri_listFileFromDirectory';
```

```
function OpenOP: boolean; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_OpenOP';
```

```
function CloseOP: boolean; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_CloseOP';
```

```
// *****
```

```

// scrittura risultati
// *****

function openTxtFile(nometxtFile: string): boolean; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_openTxtFile';
function closeTxtFile(nometxtFile: string): boolean; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_closeTxtFile';
function writeRowTxtFile(nometxtFile: string): boolean; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_writeRowTxtFile';

function openXlsFile(nomeXlsFile: string): boolean; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_openXlsFile';
function saveCloseXlsFile(nomeXlsFile: string): boolean; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_closeXlsFile';
function writeCellXlsFile(cRow: integer; cCol: integer; valueCell: string): boolean; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_writeCellXlsFile';
function defineSheetXlsFile(sheet: integer): boolean; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_defineSheetXlsFile';
function deleteUnusedSheet(sheetList: string): boolean; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_deleteUnusedSheet';

function showXLSFile(NomeFile: string): boolean; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_MostraFile';

// *****
// Generale
// *****

function LoadModel(nomeFileModel: widestring; nomeFiletxt: widestring): boolean; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_LoadModel';
function TypeAnalysis: integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_TypeAnalysis';
function TotStep: integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_TotStep';
function FirstElement(tipologia: byte): integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_FirstElement';
function ElementType(numElement : integer): integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_ElementType';

// *****
// Nodi
// *****

function IFNode2D(ID_Node: integer): boolean; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_IFNode2D';

function TotNodes2d: integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name

```

```

'cls3muri_TotNodes2d';
function TotNodes3d: integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name
'cls3muri_TotNodes3d';

// *****
// Num. Elementi
// *****
function TotSteelWoodBeam: integer; external 'C:\Windows\System32\TreMuriOPdll.
dll' name 'cls3muri_TotSteelWoodBeam';
function TotTieRodm: integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name
'cls3muri_TotTieRodm';
function TotRCBeam: integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name
'cls3muri_TotRCBeam';
function TotWall: integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name
'cls3muri_TotWall';
function TotRCWall: integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name
'cls3muri_TotRCWall';
function TotRCWallLink: integer; external 'C:\Windows\System32\TreMuriOPdll.dll'
name 'cls3muri_TotRCWallLink';
function TotRCColumn: integer; external 'C:\Windows\System32\TreMuriOPdll.dll'
name 'cls3muri_TotRCColumn';
function TotSteelWoodColumn: integer; external 'C:\Windows\System32
\TreMuriOPdll.dll' name 'cls3muri_TotSteelWoodColumn';
function TotMasonryColumn: integer; external 'C:\Windows\System32\TreMuriOPdll.
dll' name 'cls3muri_TotMasonryColumn';

// *****
// Risultati nodi
// *****
function ResNodes3D(typeAnalysis: string; ID_Node:integer; stepN:integer): double;
external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_ResNodes3D';
function ResNodes2D(typeAnalysis: string; ID_Node:integer; stepN:integer): double;
external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_ResNodes2D';

// *****
// Elementi Muratura
// *****
function ResElementWall(typeAnalysis: string; num_Element:integer; stepN:integer):
double; external 'C:\Windows\System32\TreMuriOPdll.dll' name
'cls3muri_ResElementWall';
{
Returns Damage information:
- 0=Undamaged
- 1=Shear damage
- 2=Shear failure

```

- 3=Bending damage
- 4=Bending failure
- 5=Compression failure
- 6=Tension failure
- 7=Failure during elastic phase

}

```
function ResSteelWoodenBeam(typeAnalysis: string; num_Element:integer; stepN:
integer): double; external 'C:\Windows\System32\TreMuriOPdll.dll' name
'cls3muri_ResSteelWoodenBeam';
```

```
function ResRCBeam(typeAnalysis: string; num_Element:integer; stepN:integer):
double; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_ResRCBeam';
```

```
function ResTieRod(typeAnalysis: string; num_Element:integer; stepN:integer):
double; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_ResTieRod';
```

```
// *****
```

```
// Pilastri
```

```
// *****
```

```
function ResRCCColumn(typeAnalysis:string; num_Element:integer; stepN:integer):
double; external 'C:\Windows\System32\TreMuriOPdll.dll' name
'cls3muri_ResRCCColumn';
```

```
function ResSteelWoodColumn(typeAnalysis:string; num_Element:integer; stepN:
integer): double; external 'C:\Windows\System32\TreMuriOPdll.dll' name
'cls3muri_ResSteelWoodColumn';
```

```
function ResMasonryColumn(typeAnalysis:string; num_Element:integer; stepN:
integer): double; external 'C:\Windows\System32\TreMuriOPdll.dll' name
'cls3muri_ResMasonryColumn';
```

```
function ResRcWall(typeAnalysis:string; num_Element:integer; stepN:integer):
double; external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_ResRcWall';
```

```
function ResRcWallLink(typeAnalysis:string; num_Element:integer; stepN:integer):
double; external 'C:\Windows\System32\TreMuriOPdll.dll' name
'cls3muri_ResRcWallLink';
```

```
function ResFloor(typeAnalysis:string; num_Element:integer; stepN:integer): double;
external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_ResFloor';
```

```
// *****
```

```
// Gestione dei Risultati
```

```
// *****
```

```
function Constrained(typeAnalysis:string; num_Element:integer): boolean; external  
'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_Constrained';
```

```
function TotFloor: integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name  
'cls3muri_TotFloor';
```

```
function TotMaterials: integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name  
'cls3muri_TotMaterials';
```

```
function TotReinforcements: integer; external 'C:\Windows\System32\TreMuriOPdll.  
dll' name 'cls3muri_TotReinforcements';
```

```
function TotWalls: integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name  
'cls3muri_TotWalls';
```

```
function TotLevels: integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name  
'cls3muri_TotLevels';
```

```
function RCWall(typeAnalysis:string; num_Element:integer ): double; external 'C:  
\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_RCWall';
```

```
function RCWallLink(typeAnalysis:string; num_Element:integer ): double; external 'C:  
\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_RCWallLink';
```

```
function ElementWall(typeAnalysis:string; num_Element:integer ): double; external  
'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_ElementWall';
```

```
function RCBeam(typeAnalysis:string; num_Element:integer ): double; external 'C:  
\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_RCBeam';
```

```
function Nodes3d(typeAnalysis:string; num_Element:integer ): double; external 'C:  
\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_Nodes3d';
```

```
function Nodes2d(typeAnalysis:string; num_Element:integer ): double; external 'C:  
\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_Nodes2d';
```

```
function Nodes3dIdWalls(num_Element:integer ): TStringList; external 'C:  
\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_Nodes3dIdWalls';
```

```
function RCColumn(typeAnalysis:string; num_Element:integer ): double; external 'C:  
\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_RCColumn';
```

```
function SteelWoodColumn(typeAnalysis:string; num_Element:integer ): double;  
external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_SteelWoodColumn';
```

```
function SteelWoodenBeam(typeAnalysis:string; num_Element:integer ): double;  
external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_SteelWoodenBeam';
```

```
function TieRod(typeAnalysis:string; num_Element:integer ): double; external 'C:  
\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_TieRod';
```

```
function MasonryColumn(typeAnalysis:string; num_Element:integer ): double;  
external 'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_MasonryColumn';
```

```

function Floor(typeAnalysis:string; num_Element:integer ): double; external 'C:
\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_Floor';
function Reinforcement(typeAnalysis:string; num_Element:integer ): double; external
'C:\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_Reinforcement';
function Material(typeAnalysis:string; num_Element:integer ): double; external 'C:
\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_Material';
function MaterialName(id_Material:integer ): wideString ; external 'C:
\Windows\System32\TreMuriOPdll.dll' name 'cls3muri_MaterialName';

function QnodeLevel: integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name
'cls3muri_QnodeLevel';
function QmaxLevel: integer; external 'C:\Windows\System32\TreMuriOPdll.dll' name
'cls3muri_QmaxLevel';

// *****
// Dichiarazioni per dictionary
// *****

function LoadDictionary(pstrFileXMLName : String): boolean; external 'C:
\Windows\System32\xmlReportDll.dll' name 'rhfLoadDictionary';
function leggiXMLText(pstrNodeTag : String) : String; external 'C:
\Windows\System32\xmlReportDll.dll' name 'GetXMLText';
function ChiudiDictionary: boolean; external 'C:\Windows\System32\xmlReportDll.dll'
name 'CloseDictionary';

// *****
// Fine dichiarazione interfacce
// *****

```

4.19.2 Read 3Muri dataBase Access

Navigation: Using ET RAD > Code Samples >

Read 3Muri dataBase Access

for reading Access database you have to add 2 component in the form:

1. TAdoComnection
2. TAdoQuery

both components are in DbGo Palette

in this sample the property name of TAdoConnection is : ADO3MuriConn

in this sample the property name of TAdoQuery is : Q3Muri

in the onShow event on Main Form

```
// connect to 3Muri DB Access
dbNameToOpen := 'c:\programmi
if not connectDB(dbNameToOpen, 1,") then
begin
  screen.cursor := crDefault;
  showmessage('Errore di connessione DB >> ' + dbNameToOpen);
  //exit;
end;
```

suppose now you wants read fldStato field from tblDatiGenerali table:

```
procedure readStatofromDB(Sender: TObject);
var
  IDStato    : integer;
begin
  // open tblDatiGenerali table
  TableName := 'tblDatiGenerali';

  Q3Muri.close;
  Q3Muri.SQL.Clear;
  Q3Muri.SQL.Text:= 'SELECT * FROM [' + TableName + ']';
  Q3Muri.open;

  if not Q3Muri.eof then
    Q3Muri.first;
    begin
      AField := Q3Muri.FieldByName('fldStato')
      IDStato := AField.value;
    end;
  Q3Muri.Close;
  Q3Muri.SQL.Clear;
end;
```

```

function ConnectDB(FileName: TFileName; DBType : TDBType; pwd : string):
boolean;
var
  DBName : widestring;
begin

  ADO3MuriConn.ConnectionString:='Provider=Microsoft.Jet.OLEDB.4.0;Data Source='
+ FileName + ';Persist Security Info=False';
  if pwd <> '' then ADO3MuriConn.ConnectionString:=ADO3MuriConn.ConnectionString
+ ';Jet OLEDB:Database Password=' + pwd;
  DBName := FileName;

  ADO3MuriConn.LoginPrompt := False;

  result:=True;
try
  if NOT ADO3MuriConn.Connected then ADO3MuriConn.open()
  Result:=True;
except
  Result := false;
  //Exit;
end
end;

```

4.19.3 Replace function

Navigation: Using ET RAD > Code Samples >

Replace function

you can copy and use this function in your code if you need replace a character (or string) in a string

```
myString := 'Marco';
```

```
MyString := replace(MyString,'c','i');
```

```
myString := 'Mario';
```

```

function replace(Dest, SubStr, Str: string): string;
var
  Position: Integer;

```

```

    loop : boolean;
begin
    loop := true;
    while loop = true do
    begin
        Position := Pos(SubStr, Dest);
        if position > 0 then
        begin
            Delete(Dest, Position, Length(SubStr));
            Insert(Str, Dest, Position);
        end else loop := false;
    end;
    Result := Dest;
end;

```

4.19.4 Read and Write INI Files

Navigation: Using ET RAD > Code Samples >

Read and Write INI Files

The .INI files have a text-based file format for representing application configuration data.

Initialization or Configuration Settings file (.INI) is a text file with 64Kb limit divided into sections, each containing zero or more keys. Each key contains zero or more values. Example:

```

[SectionName]
keyname1=value
keyname2=value

```

Section names are enclosed in square brackets, and must begin at the beginning of a line. Section and key names are case-insensitive, and cannot contain spacing characters. The **key name** is followed by an equal sign ("="), optionally surrounded by spacing characters, which are ignored.

If the same section appears more than once in the same file, or if the same key appears more than once in the same section, then the last occurrence prevails.

A key can contain string, integer or boolean **value**.

ET provides the **TIniFile** class, declared in the **inifiles** unit, with methods to store and retrieve values from INI files. Prior to working with the TIniFile methods you need to create an instance of the class:

```

uses inifiles;
...
var
    IniFile : TIniFile;
begin
    IniFile := TIniFile.Create('myapp.ini') ;

```

The above code creates an IniFile object and assigns 'myapp.ini' to the only property of the class - the **FileName property** - used to specify the name of the INI file you

are to use.

Read from INI

The TIniFile class has several "read" methods. The ReadString reads a string value from a key, ReadInteger, ReadFloat and similar are used to read a number from a key. All "read" methods have a default value that can be used if the entry does not exist. For example the ReadString is declared as:

```
function ReadString(const Section, Ident, Default: String): String; override;
```

Write to INI

The TIniFile has a corresponding "write" method for each "read" method. In other words they are WriteString, WriteBool, WriteInteger, etc.

Example:

reading and writing Report3Muri.ini

```
[BASICS_DATI_GENERALI]
BASICS_PLANS=Plans
BASICS_INSPECTION=Inspection
QUALITY_TITLE=Quality of Basics
COM_REMARK=Remarks
CODE_TITLE=Code
```

Procedure ReadIniFile;

```
var
  IniET : TInifile;
  sFileIni : string;
  dirRad : string;
begin
  dirRad := ExtractFilePath(Application.ExeName);
  sFileIni := '\Report3Muri.ini'
  IniET := TiniFile.Create(dirRad + sFileIni);
  sCampo := IniET.ReadString('BASICS_DATI_GENERALI','BASICS_PLANS', '');
  // the value of sCampo become Plans
  // now change sCampo and write new value
  sCampo := 'NEW PLANS'
  IniET.WriteString('BASICS_DATI_GENERALI','BASICS_PLANS',sCampo);
  // write and release ini file
  IniET.Free;
  // now the ini files is modify as follow:
  // [BASICS_DATI_GENERALI]
  // BASICS_PLANS=NEW PLANS
end;
```

4.19.5 Dll Call

Navigation: Using ET RAD > Code Samples >

Dll Call

CallDLLRAD script sample show you how it's easy call a DLL function

this sample call **CalculateStringExpression** function in **DLLRad** stored in **c:\windows\system32**

```
{ $FORM TForm2, Unit2.sfm }
```

```
uses
```

```
Classes, Graphics, Controls, Forms, Dialogs, StdCtrls, Windows;
```

```
var dll : THandle;
```

```
resultDll : string;
```

```
{ declaration of external functions }
```

```
function cse(espressione: string): string; external 'DllRad.dll' name  
'CalculateStringExpression';
```

```
procedure Form2Close(Sender: TObject; var Action: TCloseAction);
```

```
begin
```

```
FreeLibrary(dll);
```

```
end;
```

```
procedure Form2Create(Sender: TObject);
```

```
begin
```

```
try
```

```
dll := LoadLibrary('DllRad');
```

```
except
```

```
showmessage('Load DLL error : DllRad');
```

```
end;
```

```
end;
```

```

procedure Button1Click(Sender: TObject);
begin
    resultDll := cse(ExprEdt.Text);
    edtResult.text := resultDll;

end;

```

Note : the script project is stored in c:\program files\stadata\ET\script\samples\caldllrad

4.19.6 Split function

Navigation: Using ET RAD > Code Samples >

Split function

A simple function that accepts a string and a delimiter char, splits a string into tokens (TStringList items) delimited with a char value.

```

procedure TForm1.Button1Click(Sender: TObject) ;
var
    A: TStringList;
begin
    A := TStringList.Create;
try
    Split(' ', 'your ET guide', A) ;
    ShowMessage(a[0]) ; //your
    ShowMessage(a[1]) ; //ET
    ShowMessage(a[2]) ; //guide
finally
    A.Free;
end;
end;

```

// you can copy this function and add in your code

```

function Split(StrBuf,Delimiter: string): TStringList;
begin
    MyStrList := TstringList.create;
    LoopCount := 1;
    TmpBuf := "";
repeat
    if StrBuf[LoopCount] = Delimiter then
    begin
        MyStrList.Add(TmpBuf);

```

```

    TmpBuf := "";
end else
    TmpBuf := TmpBuf + StrBuf[LoopCount];

    inc(LoopCount);
until LoopCount > Length(StrBuf);

MyStrList.Add(TmpBuf);

Result := MyStrList;
end;
```

4.19.7 DataSet

Navigation: Using ET RAD > Code Samples >

DataSet

ET included TClientDataset component as a database-independent engine.

Maybe you need some temporary dataset.

So I show how to use TClientDataset in such mode on small sample.

1. you must create a TClientDataset instance. You may do it in design-time (simply drop a component on form) or in run-time (for example, in OnCreate event of your form):

```
table := TClientDataset.Create(Application);
```

2. you must add the field definitions:

```
table.FieldDefs.Add('ID', ftInteger, 0, False);
table.FieldDefs.Add('Status', ftString, 10, False);
table.FieldDefs.Add('Created', ftDate, 0, False);
table.FieldDefs.Add('Volume', ftFloat, 0, False);
```

3. create a dataset with specified structure:

```
table.CreateDataset
```

4. open a dataset

```
table.Open
```

5. it's all! Now you may add/edit/delete records, change an order (sort) and any another action that is available for any dataset.

For example, to add random values to records:

```
for i := 1 to 100 do
```

```
  begin
```

```
    table.Append;
```

```
    table.FieldName('ID').AsInteger := i;
```

```
    table.FieldName('Status').AsString := 'Code'+IntToStr(i);
```

```
    table.FieldName('Created').AsDateTime := Date();
```

```
    table.FieldName('Volume').AsFloat := Random(10000);
```

```
    table.Post;
```

```
end;
```

6. if you want to change an order for records, simply change IndexFieldNames property. For example, next command will sort your memory dataset by Created field:

```
table.IndexFieldNames := 'Created';
```

7. note that TClientDataset also allow to save memory dataset to file and load from file:

```
table.SaveToFile('c:\mem.cds');  
table.LoadFromFile('c:\mem.cds');
```

4.20 Sample Script

Navigation: Using ET RAD >

Sample Script

in directory Demos there are 5 script for an overview how to manage code:

Demos\Access3Mreader\helloWorls.ssproj

A simple script for understand how to manage components

Demos\Access3Mreader\snake.ssproj

A simple game for understand how to manage components and methods

Demos\Access3Mreader\Access3M.ssproj

learn how to interface 3Muri Access dataBase.

\Demos\db\DBCcustomers

learn how to use Data Base components.

Demos\StringGrid\StringGridSample.ssproj

learn how to use StringGrid component. Insert e read values with string grids

very important Note: when you design a stringGrid in the form the component has the property write on cells automatically set off.

this is a misteriously thing. and the property for set editing on is cached under Options

the first thing you have to do with String Grid component is :

1. Go to StringGrid property
2. Move to Options > goEditing and change False to True